UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: HEAT: Herramienta de software para la Enseñanza de Administración de Transacciones
Autores: Restelli Noelia
Director: Rodolfo Bertone
Codirector: Thomas Pablo
Asesor profesional: -
Carrera: Licenciatura en Sistemas (Plan 2003)

Resumen

El tema de transacciones, seguridad e integridad de datos es un tema importante en el área de bases de datos. Resulta interesante contar con una herramienta que permita mostrar de manera simulada el uso de transacciones para garantizar la consistencia de los datos. En esta tesina se desarrolló la *Herramienta de software para la Enseñanza de Administración de Transacciones* (HEAT), la cual permite la ejecución de modelos de simulación de transacciones de forma visual, presentando al alumno el comportamiento de la base de datos y las acciones que aseguran mantener la integridad de la información ante casos de error o fallos.
Con el uso de esta herramienta, el alumno puede definir una o varias transacciones que operarán sobre la base de datos, puede agregar situaciones de fallo de sistema o fallo en la ejecución de transacciones concurrentes, y analizar por qué la base de datos alcanza un estado de inconsistencia y, posteriormente, cómo se recupera de dicho estado.
Además, el modelo de simulación permite trabajar para entornos monousuarios con dos algoritmos alternativos: bitácora y doble paginación; para entornos concurrentes sólo fue desarrollado para ser utilizado con el algoritmo basado en bitácora.

Palabras Claves

Bases de datos, transacciones, planificaciones, protocolos de recuperación, bitácora, doble paginación, herramienta educativa.

Trabajos Realizados

Se desarrolló una herramienta educativa llamada HEAT. La misma es una aplicación Web que permite simular la utilización de transacciones en entornos monousuarios y concurrentes, permitiendo modificar parámetros, y analizar cómo los mismos pueden afectar el contenido de la base de datos.

Conclusiones

HEAT es un asistente didáctico cuyo principal objetivo es la transmisión de conceptos teóricos y prácticos en la enseñanza de transacciones. Mediante su utilización, el alumno posee una herramienta que permite comprender el funcionamiento de transacciones y cómo, mediante el uso de técnicas basadas en bitácora o doble paginación, se asegura la integridad de la información contenida en la base de datos contra problemas generados a partir del uso cotidiano de la misma.

Trabajos Futuros

- Ampliación de la herramienta para incorporar la utilización de checkpoints para los algoritmos de recuperación basados en bitácora.
- Simulación de actualizaciones sobre buffer, previo a la escritura real de la base de datos.
- Incorporación de una funcionalidad que permita inspeccionar el estado general del sistema en distintos momentos de la historia de ejecución.

Fecha de la presentación: Agosto de 2011

HEAT

Herramienta de software para la Enseñanza de Administración de Transacciones

Restelli Noelia

Facultad de Informática
Universidad Nacional de La Plata

Director: Mg. Rodolfo Bertone
Codirector: Mg. Pablo Thomas

A mis padres, por el apoyo incondicional que
me brindaron a lo largo de toda mi vida.

ÍNDICE

1. INTRODUCCIÓN

A lo largo de los últimos cincuenta años, el uso de las bases de datos creció de manera muy notoria en todas las empresas y/o corporaciones del mundo. En los inicios de la informática, muy pocas personas interactuaban directamente con los sistemas de bases de datos, hasta que la revolución de Internet a finales de la década de 1990 aumentó significativamente el acceso del usuario cada vez más convencionales a las mismas [Dat01].

Diariamente se interactúa con bases de datos, a menudo ignorando todo el procesamiento que el sistema de bases de datos desarrolla para llevar a cabo una tarea, como, por ejemplo, realizar una transferencia bancaria, consultar la disponibilidad de asientos de una empresa aérea o pagar un servicio. El conjunto de acciones que deben llevarse a cabo son realizadas por el sistema de administración de base de datos (DBMS: Data Base Management System). Un DBMS consiste en una colección de datos interrelacionados y un conjunto de programas para acceder a dichos datos. La colección de datos, normalmente denominada base de datos, contiene información relevante para una empresa. El objetivo principal de un DBMS es proporcionar una forma de almacenar y recuperar la información de una base de datos de manera que sea tanto práctica como eficiente. Los sistemas de bases de datos se diseñan para gestionar grandes cantidades de información. La gestión de los datos implica tanto la definición de estructuras para almacenar la información como la provisión de mecanismos para la manipulación de la información. Además, los sistemas de bases de datos deben proporcionar la fiabilidad de la información almacenada, a pesar de las caídas del sistema o los intentos de acceso sin autorización. Si los datos van a ser compartidos entre diversos usuarios, el sistema debe evitar posibles resultados anómalos [Sil02].

Las principales funciones de un DBMS incluyen:

- Evitar redundancia e inconsistencia de datos.
- Proveer una alta disponibilidad de los datos.
- Evitar anomalías en el acceso concurrente.
- Restringir accesos no autorizados.
- Proveer seguridad ante la presencia de fallos.
- Garantizar integridad en los datos.

Toda esta funcionalidad permanece oculta al usuario, logrando un alto nivel de abstracción en el acceso a la información.

Con el objetivo de complementar la transmisión de conceptos teóricos y prácticos en la enseñanza de transacciones de bases de datos, se diseñó HEAT: Herramienta de software para la Enseñanza de Administración de Transacciones. La misma es una herramienta educativa que

permite la ejecución de modelos de simulación de transacciones de manera visual, presentando al alumno el comportamiento de la base de datos y las acciones que aseguran la conservación de la integridad de la información ante casos de error o fallos.

Con el uso de esta herramienta, el alumno puede definir una o varias transacciones que operarán sobre la base de datos, pudiendo agregar situaciones de fallo de sistema o fallo en la ejecución de transacciones concurrentes, y analizar por qué la base de datos alcanza un estado de inconsistencia y, posteriormente, cómo se recupera de dicho estado.

Además, el modelo de simulación genera para entornos monousuarios los dos algoritmos alternativos: bitácora y doble paginación. Para entornos concurrentes sólo fue implementado el algoritmo basado en bitácora, de acuerdo a lo que utiliza la teoría de la asignatura Introducción a las Bases de Datos.

El desarrollo de una herramienta de enseñanza implica un compromiso en lo que respecta a la claridad, sencillez y robustez. Claridad, para que el usuario que recurra a la herramienta pueda afianzar los conceptos teóricos adquiridos sobre transacciones e integridad de datos, utilizando un complemento que lo ayude a afianzar sus ideas teóricas. Sencillez, para que este complemento de enseñanza, lejos de ser una complicación adicional para el alumno, sea un recurso fácil de utilizar y comprender, que motive al estudiante a recurrir a la misma. Robustez, con el objetivo de que el usuario tenga plena confianza en la herramienta.

Por ser una herramienta educativa, se intenta brindar la mayor retroalimentación posible al usuario al momento de crear los casos de prueba y simular la ejecución. La herramienta que se presenta está dirigida a usuarios con conocimientos en informática. No obstante, es objetivo primordial de la misma proveer los mecanismos necesarios para que el usuario logre generar casos de prueba correctamente y comprender el funcionamiento del tema de estudio.

A su vez, se procura que el seguimiento de la traza de ejecución logre ser claramente comprendida. HEAT provee una funcionalidad específica que permite llevar a cabo una ejecución paso a paso, destacando de manera visual las áreas de interés a las que el usuario debe prestar especial atención y exhibiendo mensajes informativos que explican la ejecución conforme se lleva a cabo, favoreciendo la correcta transmisión de conceptos.

1.1. MOTIVACIÓN

Durante la teoría de la asignatura todos los conceptos ligados con transacciones monousuarias o concurrentes son explicados en detalle con suficientes ejemplos que caracterizan un sinnúmero de casos de prueba. Sin embargo, todos los ejemplos se realizan sobre pizarrón, o eventualmente transparencias, de manera estática, sin la posibilidad de cambiar acciones o parámetros, y analizar cómo los mismos pueden afectar el contenido de la base de datos. Es por este motivo que resulta de particular interés contar con esta herramienta, permitiendo generar en tiempo real los casos de prueba y, asimismo, poder modificar los mismos para explicar diferentes situaciones al alumno.

2. TRANSACCIONES

Los sistemas de gestión de bases de datos ejecutan operaciones en unidades lógicas denominadas transacciones. Las mismas deben cumplir con un conjunto de propiedades las cuales garantizan que la base de datos se mantenga en un estado consistente conforme se ejecutan múltiples operaciones de modificación de la base de datos. Una transacción es una unidad lógica de trabajo formada por un conjunto de operaciones, donde las mismas se deben ejecutar en su totalidad o ninguna se lleva a cabo. Si una transacción no pudiera continuar con una ejecución normal, es responsabilidad del *Componente de Gestión de Recuperaciones* deshacer los cambios realizados de forma tal que el resultado sea equivalente a no haber ejecutado dicha transacción. Las operaciones que las conforman incluyen la inserción, actualización, eliminación y recuperación de información. Si las operaciones que componen una transacción solamente son operaciones de lectura, es decir, operaciones de recuperación de información, la misma se denomina *transacción de sólo lectura* [Elm02].

Una base de datos reside en almacenamiento no volátil, siendo típicamente unidades de disco. La información se lee desde la base de datos y se escribe en la misma a través de unidades de transferencia denominadas bloques. Los mismos son de tamaño fijo y pueden contener desde uno a varios sectores de disco.

Cuando una transacción comienza su ejecución se reserva un espacio en memoria principal para el mantenimiento de sus datos locales, el cual estará asignado hasta que la misma finalice su ejecución. Cuando se realiza una operación de lectura, es necesario transferir desde disco hacia memoria principal los bloques de datos que contienen la información de interés. Análogamente, para una operación de escritura se debe transferir, mediante la utilización de bloques, la información desde memoria principal a disco. La gestión de bloques, tanto para lectura como escritura, está a cargo del sistema operativo en conjunto con el DBMS.

Se consideran de relevancia las operaciones de lectura y escritura de datos, puesto que sólo interesa conocer qué datos son accedidos por las distintas transacciones. Se definen entonces las operaciones básicas de acceso a un elemento dentro de la base de datos:

Leer(X): Asigna el valor del elemento de datos *X* a la variable local x_i. Para realizar esto, debe verificarse si el bloque B_x, donde reside el elemento de datos *X*, se encuentra en memoria principal. De no ser así, debe examinarse el disco en búsqueda del mismo. Una vez encontrado, se transfiere a memoria principal el bloque B_x y luego se copia en el área de memoria local de la transacción en curso el valor de *X* a la variable local x_i.

Escribir(X): Asigna el valor de x_i al elemento de datos *X*. Nuevamente, debe verificarse que el bloque B_x se encuentre en memoria principal. En caso contrario, debe buscarse en disco y luego transferirse a memoria. Una vez localizado el bloque, se copia el valor de la variable local x_i al elemento de datos *X*, que eventualmente será actualizado en disco.

Cabe destacar que la operación de escritura no necesariamente se refleja automáticamente en almacenamiento no volátil. Debido a que un bloque tiene capacidad para almacenar muchos elementos de datos, es razonable que esta actualización no se realice cada vez que un dato es modificado, puesto que conllevaría un tiempo de sobrecarga significativo. Asimismo, tanto para la operación de lectura como de escritura, siempre debe verificarse que el bloque donde reside el elemento de datos *X* se encuentre en memoria principal, aún para la operación de escritura, la cual supone que se ha transferido previamente dicho bloque a disco. Esto es así dado que no puede asegurarse la existencia de un bloque en memoria principal a causa de las políticas de reemplazo. Comúnmente, suele utilizarse la política LRU (Least Recently Used) para realizar los intercambios de bloques entre disco y memoria, la cual consiste en reemplazar los bloques de memoria que han sido referenciados menos recientemente y llevarlos a disco con el fin de liberar espacio para nuevos bloques. El objetivo de las técnicas de sustitución de bloques de memoria intermedia es minimizar la cantidad de accesos a disco.

2.1. ESTADOS DE UNA TRANSACCIÓN

Al iniciarse una transacción, esta siempre comienza en el estado *activa* y, ante la ausencia de fallos, se mantendrá en el mismo a lo largo de toda su ejecución. Una vez que la transacción ejecuta la última instrucción, pasa al estado *parcialmente comprometida*. Si bien en este punto la transacción ha culminado su ejecución, puede suceder que los datos todavía se encuentren solamente en memoria principal y no hayan sido volcados en almacenamiento no volátil. En caso de ocurrir un fallo, la transacción deberá ser abortada, por lo que pasará al estado *fallida* y finalmente al estado *abortada*. E*s* por este motivo que la transacción no pasa al estado comprometida hasta que no se asegure que se han escrito en disco los nuevos datos o, al menos, la información necesaria para poder realizar las acciones de recuperación ante la ocurrencia de un fallo. Sólo cuando puede asegurarse este requerimiento, la transacción se encuentra en condiciones de pasar al estado comprometida. Los estados posibles de una transacción se indican en el diagrama de la figura 2.1.

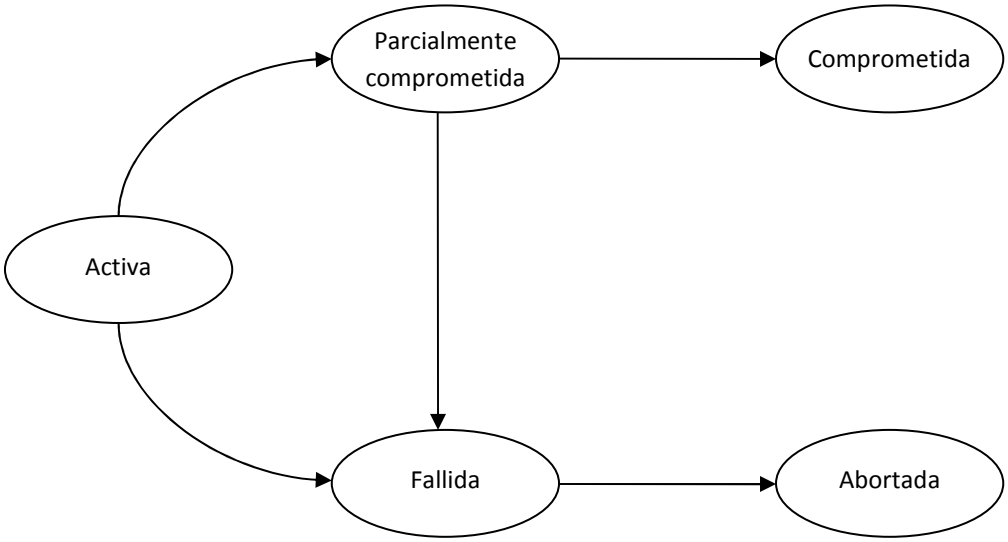


Figura 2.1

Una transacción culmina su ejecución ya sea porque finalizó exitosamente llegando a comprometerse los cambios en disco, o bien por haber sido abortada. En el último caso, la transacción puede ser reiniciada sólo en caso de que el fallo se haya producido por un error ajeno a la propia lógica de la transacción. De no ser así, la misma deberá ser cancelada y corregida para una ejecución posterior. Si se intentara ejecutar nuevamente la misma transacción, el fallo volvería a producirse puesto que el origen del mismo prevalecería.

2.2. PROPIEDADES

Para asegurar la integridad de la base de datos, es necesario que en caso de ocurrir un fallo durante la ejecución de una transacción, los cambios realizados por la misma se deshagan de manera tal que el resultado de una ejecución fallida sea equivalente a no haber ejecutado dicha transacción. De aquí se desprende la noción de atomicidad, la cual establece que, o bien todas las operaciones de una transacción se llevan a cabo, o ninguna lo hace.

Se propone como ejemplo la transacción T_0 la cual se encarga de transferir fondos entre dos cuentas bancarias. La misma se define en la figura 2.2.1.a. Un fallo luego de ejecutar la operación Escribir(A), dejaría a la cuenta *A* con el monto correspondiente a una transferencia exitosa, cuando realmente nunca llegó a depositarse el monto en la cuenta *B*.

Se considera una nueva implementación de la transacción T_0', donde primero se deposita el monto en la cuenta *B* y luego se debita el saldo de la cuenta *A*, como se ilustra en la figura 2.2.1.b.

T_0	T_1
Leer(A)	Leer(B)
A := A - 100.000	B := B + 100.000
Escribir(A)	Escribir(B)
Leer(B)	Leer(A)
B := B + 100.000	A := A - 100.000
Escribir(B)	Escribir(A)
a.	**b.**

Figura 2.2.1

De manera similar, el resultado de la ocurrencia de un fallo luego de la operación Escribir(B) dejaría a la cuenta *B* con el valor correspondiente a una transacción exitosa, mientras que en la cuenta *A* no se percibiría modificación alguna.

En la primera implementación de la transacción, para el cliente de la cuenta A *desaparece* el monto a transferir, dado que en la cuenta destino nunca se ve reflejada la transferencia, mientras que en la segunda implementación *aparece* el monto correspondiente a la transferencia cuando el mismo nunca fue deducido de la cuenta origen. En ambos casos, el resultado de la ejecución depende de la implementación y además conduce a un resultado erróneo, dejando a la base de datos en un estado inconsistente. Es por eso que, ante la eventual ocurrencia de fallos durante la ejecución de una transacción, es responsabilidad del *Componente de Gestión de Recuperaciones* llevar la base de datos a un estado consistente previo a la ocurrencia del mismo.

Para evitar este tipo de inconvenientes es necesario que las transacciones cumplan con una serie de propiedades, las cuales en conjunto aseguran la consistencia e integridad de la base de datos, aún cuando una transacción no pudiera completarse exitosamente. Dichas propiedades son:

- Atomicidad: Esta propiedad garantiza que se ejecutan todas las operaciones que conforman una transacción en su totalidad o ninguna de ellas lo hace. Es decir, se ejecutan de manera atómica.
- Consistencia: Garantiza que la base de datos queda en un estado consistente luego de la ejecución de una transacción.
- Aislamiento: Asegura que la ejecución concurrente de transacciones no interfiere en la ejecución de una transacción.
- Durabilidad: Garantiza que los cambios efectuados por una transacción persistan en la base de datos una vez que esta finaliza su ejecución, aun ante la posible ocurrencia de fallos luego de la finalización de la misma.

Estas propiedades a menudo suelen llamarse *propiedades ACID*, por sus siglas en Inglés (Atomicity, Consistency, Isolation, Durability respectivamente).

A continuación, se describe cada una de estas propiedades a través de un ejemplo aplicado, tomando como base la transacción definida en la figura 2.2.1.a.

Suponga que se produce un fallo luego de realizar la operación Escribir(A). La propiedad de atomicidad asegura que, una vez que el sistema se reanude, se deshagan todos los cambios realizados por la transacción al punto previo al inicio de su ejecución. De este modo, se garantiza que la transacción tenga el efecto de nunca haberse ejecutado. También se mantiene la consistencia, puesto que de no deshacer los cambios, desaparecería saldo de la cuenta *A* sin que se deposite en la cuenta destino, provocando una inconsistencia en la base de datos. Cabe destacar que la propiedad de consistencia queda en manos del programador de la transacción. Por ejemplo, si en la transacción se omitiera la sección donde se transfiere el monto a la cuenta destino, se violaría la propiedad de consistencia. De este modo, la responsabilidad no recae en el DBMS, sino en el programador de la transacción [Ham06].

Para ilustrar el concepto de aislamiento, suponga que inmediatamente después de la operación Leer(A) de T_0, otra transacción T_1 realiza la misma operación. Cada transacción dispondrá una copia de *A* en su espacio de almacenamiento local. El problema surge cuando ambas transacciones modifican dicho valor y luego lo escriben. El valor final de *A* corresponderá a la transacción que realice la última operación de escritura. La base de datos permanecerá en un estado inconsistente aunque ambas transacciones finalicen correctamente, dado que el valor leído por T_1 no es un dato válido por encontrarse sujeto a modificaciones. La propiedad de aislamiento garantiza que no ocurra este tipo de interferencia en la ejecución concurrente de transacciones.

Cabe destacar que no necesariamente siempre que una transacción termina correctamente se escriben sus cambios inmediatamente en disco. Por lo tanto, dichas modificaciones permanecen en memoria principal hasta que se guarden en almacenamiento no volátil. En caso de fallo, el contenido de la memoria se perderá, aún cuando las transacciones se hayan comprometido. La propiedad de durabilidad garantiza que siempre que una transacción finalice exitosamente, sus cambios serán reflejados en almacenamiento no volátil, aun ante la eventual ocurrencia de fallos de cualquier índole. La misma se garantiza mediante el mantenimiento de un registro histórico o bitácora, cuyo objetivo es generar y almacenar la información necesaria para poder llevar a cabo una recuperación de la base de datos ante la eventual ocurrencia de fallos.

2.3. TIPOS DE FALLOS

Existen numerosos tipos de fallos con los que puede encontrarse una transacción a lo largo de su ejecución. Es por este motivo que resulta de suma importancia contar con un sistema de recuperación. Los fallos pueden clasificarse de la siguiente manera:

- Fallo en la transacción: Puede ocurrir por condiciones internas de la transacción, como una división por cero, desbordamiento de un número, valores de entrada incorrectos, un error lógico en la transacción debido a una programación incorrecta o inclusive puede suceder que el usuario deliberadamente interrumpa la ejecución de una transacción. También puede ocurrir que se llegue a una situación de deadlock, por lo que una de las transacciones involucradas en el mismo deberá ser abortada para luego volver a ejecutarse.
- Caída del sistema: Existen diversos motivos por los que puede ocurrir una caída del sistema. La misma puede deberse a fallos existentes en el hardware subyacente, errores en el software de la bases de datos, errores de red o del sistema operativo. Este tipo de fallos ocasiona la pérdida del contenido de la memoria no volátil, dejando intacta la unidad de almacenamiento secundario. La información almacenada en disco es utilizada para recuperarse de este tipo de fallos.
- Fallo de disco: Puede suceder cuando la cabeza lectora-escritora roza el disco, resultando en un daño permanente a la información almacenada en el mismo, o bien un fallo en la transferencia de la información. La recuperación de este tipo de fallos se lleva a cabo utilizando las copias de seguridad realizadas en otros dispositivos, como discos o cintas magnéticas [Wie10].
- Contrariedades físicas: Esto incluye el corte del suministro eléctrico, catástrofes como inundaciones o incendios, robo, sabotaje, entre otras. Las acciones llevadas a cabo para recuperarse de este tipo de inconvenientes requieren de copias de seguridad realizadas en almacenamientos secundarios distribuidos físicamente [Kin90] [Hof07].

En todos los casos, es necesario contar con información de respaldo para poder restaurar la base de datos. Esta responsabilidad recae principalmente en las personas encargadas de administrar la base de datos, quienes deben realizar copias periódicas de la misma. Asimismo, el DBMS se encarga de llevar a cabo un determinado conjunto de acciones durante la ejecución normal para que, en caso de requerirse, se cuente con la información necesaria de manera tal de poder llevar la base de datos a un estado consistente previo a la ocurrencia de un fallo, sin que esto ocasione pérdida de información o inconsistencias lógicas [Kin90].

2.4. TRANSACCIONES CONCURRENTES

Los sistemas operativos hacen un uso extensivo de la multiprogramación, la cual consiste en mantener varios programas en la memoria e intercalar la ejecución de los mismos [Sil99]. Esto permite obtener un rendimiento considerablemente mayor en la utilización de la CPU, dado que la misma no queda ociosa si el programa de usuario en curso queda a la espera de, por ejemplo, una operación de entrada/salida o el ingreso de información por parte del usuario. El mismo concepto puede aplicarse a la ejecución de transacciones, donde no es necesario que una transacción finalice su ejecución para que tome lugar una nueva, dando origen a la

ejecución concurrente de transacciones. Esta técnica consiste en ejecutar las instrucciones que conforman las distintas transacciones de manera intercalada, asignando el procesador por un período determinado a cada una de las transacciones activas. El principal beneficio es un aumento en la productividad del sistema por dos razones: una transacción está formada por un conjunto de operaciones, las cuales incluyen operaciones de entrada y salida, como lectura y escritura de disco o impresión de datos. En la ejecución concurrente de transacciones, cuando una transacción queda a la espera de este tipo de operaciones, puede cederse el uso del procesador a otra transacción. Dado que la CPU y los dispositivos de entrada/salida pueden trabajar de forma independiente, se obtiene un rendimiento superior al explotar el paralelismo existente en la utilización de los distintos componentes de hardware. Esto evita esperas innecesarias y además se obtiene una mayor utilización del procesador, así como del disco, dado que ninguno permanece demasiado tiempo sin ser utilizado.

Otro factor que permite mejorar el desempeño se presenta cuando existen transacciones que necesitan un tiempo de procesamiento considerablemente mayor que las demás [Yad03]. Si las mismas se ejecutan secuencialmente, las transacciones de menor duración deben esperar a que las más prolongadas finalicen para poder ejecutarse. Esto conlleva retardos impredecibles en la ejecución de transacciones. Si las mismas actualizan distintas partes de la base de datos, es conveniente que se ejecuten concurrentemente ya que se aprovechan mejor los recursos, compartiendo ciclos de CPU y los accesos a disco.

Si bien la ejecución concurrente de transacciones tiene asociados grandes beneficios, trae aparejado un inconveniente: aun cuando las transacciones individuales sean correctas, la ejecución concurrente puede crear varios problemas de integridad y consistencia de los datos.

En las transacciones desarrolladas anteriormente no se presentan inconvenientes dado que se modifican distintos elementos de datos. El problema surge en el caso de que dos o más transacciones actualicen la misma información. Suponga que se tienen dos transacciones, donde cada una actualiza el stock de un producto de manera independiente, siendo el valor inicial del stock del producto de 100 unidades, como se ilustra en la figura 2.3:

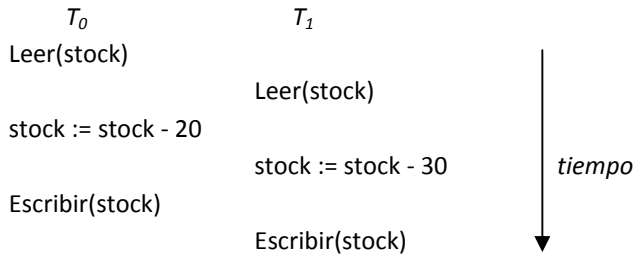


Figura 2.3

La transacción T_0 lee la variable stock y almacena el valor en su espacio de trabajo local. La transacción T_1 realiza la misma operación. En este punto, ambas transacciones han leído el mismo valor de stock, correspondiente a 100 unidades. Luego, T_0 actualiza la variable stock a 80 unidades y T_1 la actualiza a 70 unidades. Solo resta la operación de escritura por parte de ambas transacciones. En este ejemplo, primero realiza la escritura la transacción T_0 y luego T_1,

dando como resultando un valor final de stock de 70 unidades. En el caso en que las escrituras se realicen en orden inverso, el valor final de la variable stock correspondería a 80 unidades. Claramente esto supone una inconsistencia en la base de datos, dado que el valor final del stock debería ser 50, independientemente del orden de ejecución de las escrituras. Este problema se conoce como *problema de la actualización perdida*. Una solución al mismo propone que se impida que dos o más transacciones concurrentes lean el mismo elemento de datos cuando el mismo será modificado [Kro10].

Si bien ambas transacciones independientes son correctas, el orden de ejecución de las instrucciones conduce a un resultado incorrecto. Se define como *planificación* a la secuencia de instrucciones, correspondientes a distintas transacciones, a ser ejecutadas en un orden preestablecido. Las planificaciones deben conservar el orden original de ejecución de las instrucciones de cada transacción. El componente del sistema de base de datos encargado de efectuar las planificaciones es el *Componente de Control de Concurrencia*, el cual se encarga de generar solamente planificaciones que lleven la base de datos de un estado consistente a otro. De no disponerse del mismo, la planificación estaría a cargo del sistema operativo, en cuyo caso serían posibles muchas planificaciones, incluyendo aquellas que no dejan a la base de datos en un estado consistente. Es posible asegurar un estado consistente si el resultado de una ejecución concurrente es equivalente a la ejecución de las transacciones de manera secuencial.

Se presentan tres problemas que se generan a partir de planificaciones incorrectas. Se ilustran algunos de ellos con referencia a una base de datos simple para hacer reservas en una línea aérea. Se define la transacción T_0 que transfiere N reservas de un vuelo a otro, con la cantidad de lugares reservados almacenados en el elemento de datos X, para el vuelo original, e Y para el vuelo destino. Asimismo, la transacción T_1 reserva M asientos en el primer vuelo al que hace referencia la transacción T_0.

2.4.1. ACTUALIZACIÓN PERDIDA

En la figura 2.4 se ilustra una ejecución específica del programa de aplicación que se encarga de transferir N reservas y de aquel que realiza M reservas. El problema ocurre cuando distintas transacciones tienen acceso al mismo elemento de datos, y la primera transacción que accede a dicho elemento no ha sido comprometida cuando se ejecutan las subsiguientes.

En el ejemplo, ambas transacciones leen el elemento de datos X el cual contiene las reservas realizadas para un vuelo, guardan esta información en su espacio de trabajo local, lo modifican y luego escriben el cambio en la base de datos. El problema radica en que T_0 no ha sido comprometida cuando se ejecuta T_1, por lo que T_1 lee un elemento de datos inválido. Suponga que $X = 100$ antes de ejecutarse ambas transacciones, $N = 5$ (T_0 transfiere 5 reservas a Y) y $M = 10$ (T_1 realiza 10 reservas en X). El resultado final debería ser $X = 105$, pero el resultado de la ejecución deja $X = 110$, dado que se pierde la transferencia de 5 reservas al otro vuelo.

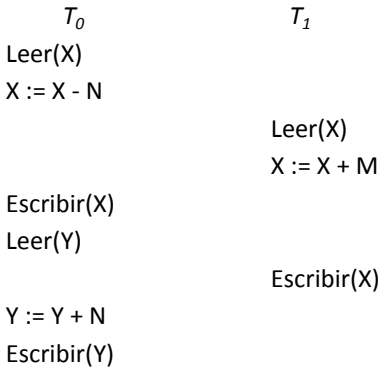


Figura 2.4

2.4.2. DATOS NO COMPROMETIDOS

El problema de datos no comprometidos surge cuando una transacción actualiza un elemento de la base de datos y luego falla. En ese momento, otra transacción tiene acceso al elemento actualizado antes de que se restaure a su valor original [Elm02].

En la figura 2.5, T_0 actualiza el elemento de datos *X* y luego falla antes de finalizar su ejecución, por lo que el proceso de recuperación restaurará el valor de *X* a su valor original. Sin embargo, T_1 alcanza a leer el valor de *X* que no será almacenado de forma permanente en la base de datos debido al fallo de T_0. Este problema también es conocido como problema de lectura sucia, dado que un dato, en este caso *X*, es creado por una transacción que todavía no se ha completado ni comprometido.

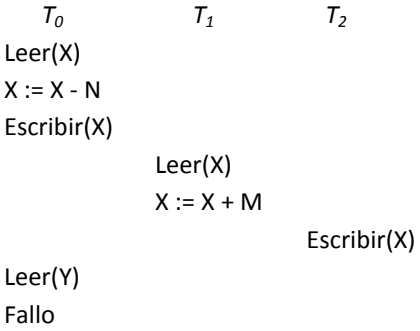


Figura 2.5

2.4.3. RECUPERACIONES INCONSISTENTES

Este problema se presenta cuando una transacción utiliza una función de agregado de resumen sobre un conjunto de datos mientras otras transacciones están actualizando la misma información. El inconveniente ocurre cuando una transacción lee algunos datos antes de ser actualizados y otros después de actualizarse, lo cual conlleva a resultados inconsistentes.

En la figura 2.6 se definen dos transacciones, donde T_0 transfiere *N* reservas a otro vuelo, y T_1 calcula el total de reservas para todos los vuelos. El número total de reservas que obtiene T_1 es incorrecto, dado que lee el valor de *X* luego de decrementar *N* reservas y lee el valor de *Y* antes de incrementarse. Por lo tanto, el resultado obtenido por la transacción T_1 será erróneo por una cantidad *N* de reservas.

T_0	T_1
	Suma := 0
	Leer(A)
	Suma := Suma + A
	.
	.
	.
Leer(X)	
X := X - N	
Escribir(X)	
	Leer(X)
	Suma := Suma + X
	Leer(Y)
	Suma := Suma + Y
Leer(Y)	
Y := Y + N	
Escribir(Y)	

Figura 2.6

Existe otro problema denominado *lectura no repetible*, en el cual una transacción lee el mismo elemento de datos en más de una oportunidad, y entre esas lecturas, otra transacción modifica dicho elemento. Esto resulta en la obtención de distintos valores para la lectura del mismo dato. Por ejemplo, esto puede ocurrir si un cliente consulta la disponibilidad de asientos en varios vuelos y, cuando decide el vuelo sobre el cual realizar la reserva, la transacción vuelve a leer el número de asientos disponibles para el vuelo seleccionado y este ha sido modificado por otra transacción.

3. PLANIFICACIONES

En un entorno que admite varias transacciones ejecutándose simultáneamente, es necesario establecer un criterio de ejecución para que la concurrencia no afecte la integridad en la base de datos. La idea subyacente en la ejecución concurrente de transacciones es llevar la base de datos de un estado consistente a otro. Considere las planificaciones de la figura 3.1 donde la base de datos alcanza tal estado.

T_0	T_1
Leer(X)	
X := X - 3000	
Escribir(X)	
Leer(Y)	
Y := Y + 3000	
Escribir(Y)	
	Leer(Y)
	temp := Y * 0,15
	Y := Y - temp
	Escribir(Y)
	Leer(X)
	X := X + temp
	Escribir(X)

a. Planificación P_0

T_0	T_1
	Leer(Y)
	temp := Y * 0,15
	Y := Y - temp
	Escribir(Y)
	Leer(X)
	X := X + temp
	Escribir(X)
Leer(X)	
X := X - 3000	
Escribir(X)	
Leer(Y)	
Y := Y + 3000	
Escribir(Y)	

b. Planificación P_1

Figura 3.1

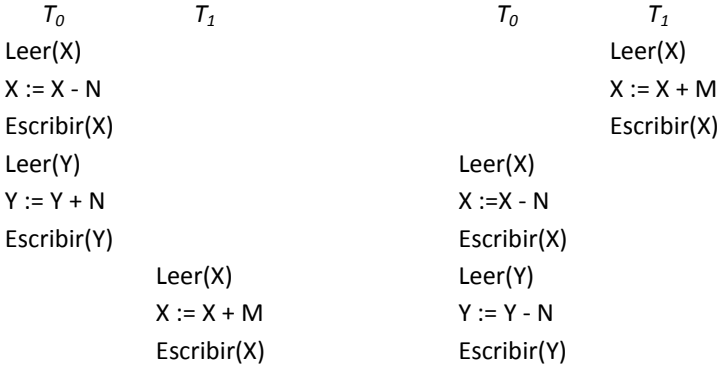
En un entorno concurrente, el orden de ejecución en serie de transacciones se denomina planificación. Así, la figura 3.1 presenta dos planificaciones diferentes. Cabe destacar que si se disponen dos transacciones es posible generar dos planificaciones en serie. En el caso que se ejecutaran tres transacciones (T_0, T_1 y T_2) entonces se podrían generar seis planificaciones en serie diferentes: < T_0, T_1, T_2 >, < T_0, T_2, T_1 >, < T_1, T_0, T_2 >, < T_1, T_2, T_0 >, < T_2, T_0, T_1 >, < T_2, T_1, T_0 >. En general, un entorno concurrente con N transacciones en ejecución puede generar $N!$ planificaciones en serie diferentes.

Una planificación debe involucrar todas las instrucciones que conforman las transacciones y además estas transacciones deben conservar el orden de ejecución definido previamente. Se puede deducir que, una vez iniciada una transacción, ninguna otra puede comenzar su ejecución en tanto la primera no haya finalizado. Esta última característica garantiza la secuencialidad en la ejecución y, por consiguiente, el cumplimiento de la propiedad de aislamiento para una transacción. El inconveniente con esta aproximación es que todas las ventajas de un entorno concurrente no pueden ser aprovechadas y el procesamiento secuencial de N transacciones demora mucho más tiempo [Ber11] [Web01].

3.1. PLANIFICACIONES EN SERIE

Una planificación se denomina *planificación en serie* si las operaciones de cada transacción se ejecutan de manera consecutiva, sin operaciones intercaladas de otra transacción. Es decir, hasta que no se completa la ejecución de una transacción, no tiene lugar otra transacción de la planificación. Por lo tanto, en una planificación en serie, sólo una transacción está activa a la vez. Sólo cuando la transacción activa se compromete o aborta, se inicia la ejecución de la siguiente transacción. Cuando las transacciones se ejecutan concurrentemente no puede asegurarse que la planificación sea en serie.

En la figura 3.2.a se define una planificación en serie formada por dos transacciones, en donde primero se ejecuta completamente T_0 y luego T_1, y en la figura 3.2.b primero se ejecuta T_1 de manera completa y luego T_0.



a. Planificación P_0 **b. Planificación P_1**

Figura 3.2

Si se consideran las transacciones como independientes, puede suponerse entonces que toda planificación en serie es correcta, dado que una transacción es correcta si se ejecuta por sí sola, por la propiedad de aislamiento. Es por este motivo que no importa el orden en el que se ejecutan las transacciones, al no haber interferencia de operaciones entre las mismas, siempre se llegará a un resultado final correcto en la base de datos. En caso de producirse un error en la ejecución de una transacción, es la propiedad de atomicidad la que garantiza la consistencia de la base de datos, como ha sido previamente desarrollado.

Una de las limitaciones de este tipo de planificaciones es que acotan el grado de concurrencia que podría ser explotado. Por ejemplo, podría ocurrir que una transacción quede a la espera de una operación de E/S, tiempo que la CPU queda ociosa pudiendo ser aprovechado por otra transacción. Inclusive ante la presencia de una transacción muy prolongada, el resto de las transacciones deberán esperar indefectiblemente a que la misma finalice a fin de lograr ejecutarse. Por tales motivos, las planificaciones en serie generalmente se consideran muy poco eficientes y por este motivo se minimiza su existencia en la práctica [Elm02].

3.2. PLANIFICACIONES EN ENTORNOS CONCURRENTES

Las planificaciones para un entorno concurrente no deben ser necesariamente planificaciones en serie. Cuando la ejecución de una transacción no afecta el desempeño de otra transacción sobre la base de datos, ambas pueden ejecutarse concurrentemente sin afectar la propiedad de aislamiento. Suponga que el cliente 59.898 del banco abona el servicio de telefonía celular y al mismo tiempo el cliente 15.233 decide abonar el servicio de gas. Se generan entonces dos transacciones: T_0 y T_1. T_0 actúa sobre las cuentas del cliente 59.898 y del prestatario de telefonía celular, mientras que T_1 actúa sobre las cuentas del cliente 15.233 y el prestatario de gas. Estas transacciones, dado que operan sobre subconjuntos de datos diferentes, no generan pérdida de consistencia en un entorno libre de fallos. El aislamiento se garantiza porque actúan sobre datos diferentes [Ber11].

Si T_0 y T_1 se ejecutan en serie se garantiza la propiedad de aislamiento. Sin embargo, se ha hecho referencia a la necesidad de ejecutar ambas transacciones de manera concurrente. Analice las planificaciones de la figura 3.3.

T_0	T_1
Leer(X)	
X := X - 3000	
Escribir(X)	
	Leer(Y)
	temp := Y * 0,15
	Y := Y - temp
	Escribir(Y)
Leer(Y)	
Y := Y + 3000	
Escribir(Y)	
	Leer(X)
	X := X + temp
	Escribir(X)

a. Planificación P_0

T_0	T_1
	Leer(Y)
	temp := Y * 0,15
	Y := Y - temp
	Escribir(Y)
Leer(X)	
X := X - 3000	
Escribir(X)	
	Leer(X)
	X := X + temp
	Escribir(X)
Leer(Y)	
Y := Y + 3000	
Escribir(Y)	

b. Planificación P_1

Figura 3.3

En la planificación P_0 primero se ejecutan las tres instrucciones iniciales de T_0, luego las cuatro instrucciones iniciales de T_1, siguiendo con las últimas instrucciones de T_0 y luego las de T_1. Como se asume un entorno de ejecución libre de fallos, se conserva la integridad de la base de datos. Esta es una planificación concurrente que lleva la base de datos de un estado consistente a otro estado consistente, de la misma forma que P_1.

Considere ahora las planificaciones ilustradas en la figura 3.4. Estas planificaciones no mantienen la integridad de la base de datos. De aquí puede concluirse que no todas las planificaciones concurrentes son válidas. La inconsistencia temporal puede ser causa de inconsistencia en planificaciones concurrentes. La misma se presenta desde el momento en

que una transacción modifica la base de datos hasta que termina, comprometiendo su ejecución, o bien abortando.

T_0	T_1
Leer(Y)	
	Leer(Y)
	temp := Y * 0,15
	Y := Y - temp
Y := Y + 3000	
Escribir(Y)	
	Escribir(Y)
	Leer(X)
Leer(X)	
	X := X + temp
	Escribir(X)
X := X - 3000	
Escribir(X)	

a. Planificación P_0

T_0	T_1
	Leer(Y)
Leer(Y)	
Y := Y + 3000	
	temp := Y * 0,15
	Y := Y - temp
	Escribir(Y)
	Leer(X)
Escribir(Y)	
Leer(X)	
	X := X + temp
X := X - 3000	
Escribir(X)	
	Escribir(X)

b. Planificación P_1

Figura 3.4

3.3. RECUPERABILIDAD DE PLANIFICACIONES

Debe tenerse en cuenta que cuando una transacción falla, por cualquiera de los motivos mencionados previamente en este texto, es necesario deshacer los cambios realizados por dicha transacción de forma tal de asegurar las propiedades de atomicidad y consistencia. A su vez, también debe asegurarse que toda transacción T_j que lea datos previamente escritos por una transacción T_i, también se aborte. Entonces, es importante reconocer aquellas relaciones de dependencia existentes entre distintas transacciones.

De aquí se deriva la noción de planificaciones recuperables y no recuperables. Las *planificaciones recuperables* una vez comprometidas no necesitarán deshacer sus cambios (*rollback*). Para esto, es necesario definir cuándo existe una relación de dependencia entre transacciones. La dependencia existe si para todo par de transacciones T_i y T_j, tales que T_j lee elementos de datos que previamente ha escrito T_i, la operación *commit* T_i aparece antes que la de T_j. A diferencia de las *planificaciones no recuperables* donde esta restricción no puede ser garantizada, por lo que este tipo de planificaciones no deberían ser permitidas por el DBMS [Sil02].

En la figura 3.5 se define la transacción T_0 que actualiza el valor de X y la transacción T_1 que luego accede a este valor y se compromete antes que T_0 lo haga. Si T_0 aborta, el valor de X será restaurado a su antiguo valor. El problema surge al momento de deshacer T_1, ya que la misma

ha alcanzado el estado *comprometida* y, por lo tanto, no es posible retroceder dicha transacción, por la propiedad de durabilidad. De esta forma, se llega una situación en la cual no es posible recuperarse correctamente del fallo de T_0.

T_0	T_1
Leer(X)	
X := X * 0,5	
Escribir(X)	
	Leer(X)
Leer(Y)	

Figura 3.5

3.4. EQUIVALENCIA DE PLANIFICACIONES

Para comprender el significado de equivalencia de planificaciones, sólo se tendrán en cuenta las operaciones de lectura y escritura, puesto que estas son las únicas operaciones que acceden o modifican el contenido de la base de datos. El resto de las operaciones actúan de manera local sobre la memoria volátil y, por consiguiente, no alteran el contenido de la base de datos.

Analice el ejemplo de la figura 3.6, donde ambas planificaciones producen el mismo estado final de la base de datos. Estas planificaciones producen el estado final por azar en el caso que el valor inicial de *X* = 150, pero para otros valores de *X* no produce un resultado correcto, por lo que puede concluirse que las planificaciones no son equivalentes. Este tipo de equivalencia se denomina *equivalencia por resultados*, puesto que sólo compara los efectos finales de las planificaciones sobre la base de datos. La equivalencia por resultados no representa un análisis viable para la ejecución de transacciones. La ejecución consistente de transacciones no puede estar supeditada a los datos sino a la forma en que se ejecuta.

Leer(X)	Leer(X)
X := X * 2	X := X + 150
Escribir(X)	Escribir(X)
a. Planificación P_1	**b. Planificación P_2**

Figura 3.6

Como se ha visto, este tipo de equivalencia no es aceptable. El enfoque más seguro y general para definir la equivalencia de dos planificaciones consiste en no hacer suposiciones sobre el tipo de operaciones que intervienen en las transacciones. Se utiliza el concepto de seriabilidad de planificaciones para identificar qué planificaciones son correctas cuando existe

intercalación de operaciones. Más adelante se introducen los conceptos de seriabilidad en cuanto a conflictos y seriabilidad en cuanto a vistas.

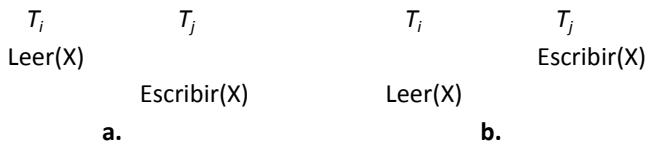
3.5. CONFLICTOS EN PLANIFICACIONES CONCURRENTES

Dada una planificación en la cual hay dos instrucciones consecutivas, pertenecientes a distintas transacciones, si estas instrucciones acceden a diferentes elementos de datos pueden intercambiar su orden de ejecución sin afectar el resultado final de la base de datos, independientemente del tipo de operación que se ejecute. En cambio, si ambas transacciones acceden al mismo elemento de datos y al menos una de estas operaciones es una operación de escritura, entonces el orden en que se ejecutan estas instrucciones es significativo puesto que puede conducir a diferentes resultados.

Se detallan a continuación las distintas combinaciones de operaciones posibles, indicando cuáles son las implicaciones en cada caso. Para esto, se definen dos transacciones, T_i y T_j, donde ambas acceden al elemento de datos *X*. Para indicar a qué transacción pertenece cada operación, se utilizará la notación Leer(X)$_T$ o Escribir(X)$_T$ según corresponda.

1. Leer(X)$_{Ti}$, Leer(X)$_{Tj}$: En el caso en que ambas operaciones sean operaciones de lectura, el orden de ejecución de las mismas es indistinto, puesto que no se está modificando el valor de *X* en la base de datos. Por ende, ambas instrucciones recuperan el mismo valor de *X*.

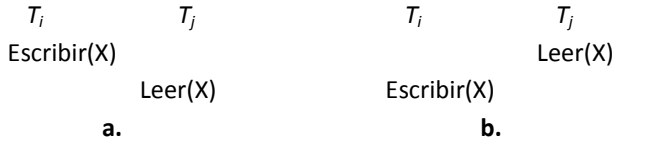
2. Leer(X)$_{Ti}$, Escribir(X)$_{Tj}$: En este caso, pueden presentarse dos alternativas en el orden de ejecución de las operaciones:



a. **b.**

   El orden de ejecución tiene importancia, dado que en el caso (a) la transacción T_i obtiene el valor de *X* antes de ser modificado, mientras que en el caso (b) la misma transacción obtiene un valor de *X* diferente, puesto que es accedido por T_i luego de ser modificado.

3. Escribir(X)$_{Ti}$, Leer(X)$_{Tj}$: Como en el caso anterior, pueden presentarse dos alternativas:



a. **b.**

Nuevamente, el orden tiene importancia, dado que en (a) la transacción T_j obtiene el valor de *X* luego de ser modificado, mientras que en (b) la misma transacción obtiene un valor de *X* diferente, puesto que el elemento de datos *X* es leído antes de realizar la modificación.

4. Escribir(X)$_{Ti}$, Escribir(X)$_{Tj}$: Al no estar involucrada ninguna operación de lectura, el orden no afecta a ninguna de las transacciones involucradas, pero sí afecta a la próxima instrucción Leer(X) que se realice, ya que la base de datos conservará el valor de *X* correspondiente a la última operación Escribir(X). Puede concluirse que el orden en que se ejecutan dichas operaciones es sumamente importante, pues afecta directamente el valor final de *X* en el estado de la base de datos.

A partir de los casos posibles que pueden conducir a inconsistencias sobre la base de datos, se definen las condiciones necesarias que deben cumplirse para que dos operaciones se encuentren en conflicto:

- Las operaciones deben pertenecer a distintas transacciones.
- Las operaciones deben acceder al mismo elemento de datos.
- Al menos una de las operaciones debe ser una operación de escritura.

Para ilustrar el concepto de conflicto entre instrucciones, considere la planificación de la figura 3.7, donde se definen dos transacciones que presentan operaciones conflictivas, puesto que ambas operan sobre el elemento de datos *X* con operaciones de lectura y escritura.

T_0	T_1
Leer(X)	
	Leer(X)
X := 0	
Escribir(X)	
Leer(Y)	
	X := X + 1000
	Escribir(X)
	Leer(Y)
Leer(Z)	
Z := Z * 0,10	
Escribir(Z)	

Figura 3.7

Los conflictos existentes son:

- Leer(X) de T_0 y Escribir(X) de T_1.
- Leer(X) de T_1 y Escribir(X) de T_0.

- Escribir(X) de T_0 y Escribir(X) de T_1.

En el caso de las operaciones Leer(Y) no existe conflicto alguno dado que ambas transacciones realizan exclusivamente una operación de lectura. Las operaciones Leer(Z) y Escribir(Z) tampoco presentan conflicto pues pertenecen a la misma transacción.

3.6. EQUIVALENCIA EN CUANTO A CONFLICTOS

Sean I_i e I_j instrucciones consecutivas de una planificación *P*. Si I_i e I_j son instrucciones de transacciones diferentes y además I_i e I_j no se encuentran en conflicto, entonces puede cambiarse el orden de I_i e I_j para obtener una nueva planificación *P'*. Lo esperado es que *P* sea equivalente a *P'*, ya que todas las instrucciones aparecen en el mismo orden en ambas planificaciones, salvo I_i e I_j, cuyo orden no importa. Por consiguiente, si una planificación *P* puede transformarse en otra planificación *P'* mediante una serie de intercambios, entonces puede afirmarse que *P* y *P'* son equivalentes en cuanto a conflictos [Sil02].

En la figura 3.8.a se presenta una planificación *P* a partir de la cual se llegará a una planificación *P'* equivalente en cuanto a conflictos mostrando los intercambios necesarios para llegar a tal resultado. Como se ha mencionado previamente, puesto que solo interesan las operaciones de lectura y escritura, en la figura 3.8.b se dispone la planificación simplificada sobre la cual se realizarán los intercambios de instrucciones no conflictivas.

T_0	T_1
Leer(A)	
A := A + 10	
Escribir(A)	
	Leer(B)
	B := B + 100
	Escribir(B)
	A := 0
	Escribir(A)
Leer(B)	
B := B * 10	
Escribir(B)	

a. Planificación P

T_0	T_1
Leer(A)	
Escribir(A)	
	Leer(B)
	Escribir(B)
	Escribir(A)
Leer(B)	
Escribir(B)	

b. Planificación P simplificada

Figura 3.8

Los intercambios a realizar se ilustran en la figura 3.9. El primer intercambio a realizar consiste en la instrucción Escribir(A) de T_0 y la instrucción Leer(B) de T_1, puesto que no se encuentran en conflicto, pueden intercambiarse generando la planificación equivalente que se muestra en la figura 3.9.a. Luego, es posible intercambiar las instrucciones Leer(A) de T_0 y Leer(B) de T_1, dado que ambas son operaciones de lectura, como se muestra en la figura 3.9.b. Se continúa

con el intercambio de la instrucción Escribir(A) de T_0 con Escribir(B) de T_1, ya que ambas modifican distintos elementos de datos, obteniéndose la planificación P_3 que se ilustra en la figura 3.9.c.

Las instrucciones Leer(A) de T_0 y Escribir(B) de T_1 tampoco presentan conflicto, por lo cual es posible intercambiarlas, como se muestra en la figura 3.9.d. Cabe destacar que no es posible intercambiar las operaciones Escribir(A) de T_0 y T_1, puesto que acceden al mismo elemento de datos y ambas son operaciones de escritura. Por lo cual el próximo intercambio posible consiste en intercambiar Leer(B) de T_0 con Escribir(A) de T_1, tal cual se ilustra en la figura 3.9.e, y finalmente intercambiar Escribir(B) de T_0 con Escribir(A) de T_1, como se ilustra en la figura 3.9.f obteniéndose finalmente la planificación *P'*.

T_0	T_1
Leer(A)	
	Leer(B)
Escribir(A)	
	Escribir(B)
	Escribir(A)
Leer(B)	
Escribir(B)	

a. Planificación P_1

T_0	T_1
	Leer(B)
Leer(A)	
Escribir(A)	
	Escribir(B)
	Escribir(A)
Leer(B)	
Escribir(B)	

b. Planificación P_2

T_0	T_1
	Leer(B)
Leer(A)	
	Escribir(B)
Escribir(A)	
	Escribir(A)
Leer(B)	
Escribir(B)	

c. Planificación P_3

T_0	T_1
	Leer(B)
	Escribir(B)
Leer(A)	
Escribir(A)	
	Escribir(A)
Leer(B)	
Escribir(B)	

d. Planificación P_4

T_0	T_1
	Leer(B)
	Escribir(B)
Leer(A)	
Escribir(A)	
Leer(B)	
	Escribir(A)
Escribir(B)	

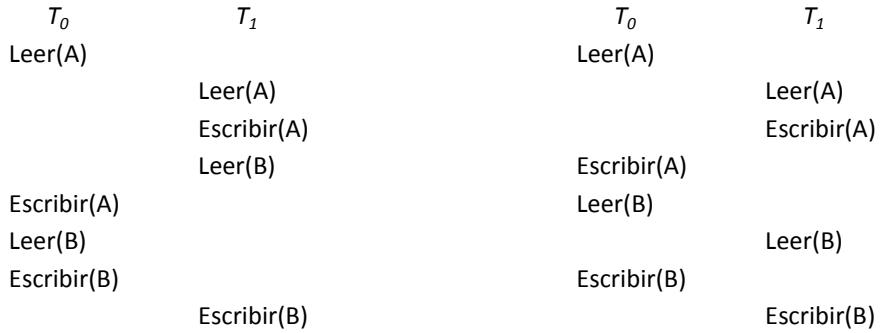
e. Planificación P_5

T_0	T_1
	Leer(B)
	Escribir(B)
Leer(A)	
Escribir(A)	
Leer(B)	
Escribir(B)	
	Escribir(A)

f. Planificación P'

Figura 3.9

El resultado final de estos intercambios conduce a una planificación *P'* la cual es equivalente por conflictos a la planificación original *P*. Considere la planificación presentada en la figura 3.10.a, en donde se aplican los intercambios correspondientes a la planificación P_r obteniéndose la planificación P_r' (figura 3.10.b). Si bien la planificación P_r' es equivalente en cuanto a conflictos con la planificación P_r, como P_r no es correcta, P_r' tampoco lo será. Esto se debe a que ninguna de estas planificaciones es equivalente a una planificación serializable.



a. Planificación P_r **b. Planificación P_r'**

Figura 3.10

3.7. SERIABILIDAD EN CUANTO A CONFLICTOS

Para asegurar que una planificación *P'* equivalente por conflictos a una planificación *P* además sea correcta, la misma debe ser serializable en cuanto a conflictos. Esto quiere decir que, luego de aplicar los intercambios entre operaciones no conflictivas, la planificación resultante *P'* debe ser equivalente a una planificación en serie. Es decir, una planificación *P* es serializable en cuanto a conflictos si es equivalente por conflictos a alguna planificación en serie *P'* [Han97].
Considere el ejemplo de la figura 3.11.a, en donde se define la planificación *P* y su correspondiente planificación serializable por conflictos *P'* en la figura 3.11.b.

a. Planificación P **b. Planificación en serie P'**

Figura 3.11

Puesto que no siempre es posible obtener este tipo de planificaciones, puede suceder que una planificación sea equivalente en cuanto a conflictos pero no serializable en cuanto a conflictos. Por ejemplo, en la figura 3.12 no puede obtenerse una planificación serializable por conflictos puesto que las instrucciones Leer(B) de T_0 y Escribir(B) de T_1 se encuentran en conflicto, impidiendo que se coloquen todas las instrucciones pertenecientes a T_0 antes de las

instrucciones de T_1. Sin embargo, los valores finales de *A* y *B* son los mismos luego de ejecutar tanto esta planificación, como las planificaciones secuenciales <T_0, T_1> o <T_1, T_0>.

T_0	T_1
Leer(A)	
Escribir(A)	
	Leer(B)
	Escribir(B)
Leer(B)	
Escribir(B)	
	Leer(A)
	Escribir(A)

Figura 3.12

3.8. EQUIVALENCIA EN CUANTO A VISTAS

La seriabilidad en cuanto a vistas es menos restrictiva que la seriabilidad en cuanto a conflictos. Al igual que en la equivalencia en cuanto a conflictos, la planificación resultante debe estar formada por el mismo conjunto de transacciones y las mismas operaciones que la planificación original. Dos planificaciones *P* y *P'* son equivalentes en cuanto a vistas si se cumplen las siguientes tres condiciones:

1. Para todo elemento de datos *X*, si la transacción T_i lee el valor inicial de *X* (es decir, lo lee desde la base de datos), entonces T_i también debe leer el valor inicial de *X* en la planificación *P'*.
2. Para todo elemento de datos *X*, si la transacción T_i ejecuta Leer(X) en *P* y el valor de *X* fue escrito previamente por una transacción T_j, entonces en la planificación *P'* la transacción T_i también debe leer el valor de *X* que haya producido la transacción T_j.
3. Para todo elemento de datos *X*, la transacción que realice la última operación de escritura en la planificación *P*, debe también realizar la última operación de escritura en la planificación *P'*.

La idea subyacente de la equivalencia en cuanto a vistas se basa en que siempre que una transacción lea el resultado de la misma operación de escritura en ambos planes, las planificaciones producirán los mismos resultados. Es por este motivo que las operaciones de lectura perciben la misma vista en ambas planificaciones, garantizado por las condiciones 1 y 2. La condición 3, en conjunto con las condiciones anteriores, asegura que ambas planificaciones dan como resultado el mismo estado final del sistema.

En la figura 3.13 se definen tres planificaciones para ilustrar el concepto de equivalencia en cuanto a vistas. Las planificaciones P_0 y P_1 no son equivalentes en cuanto a vistas puesto que la transacción T_1 no lee el mismo valor de *A* que en la planificación P_0, al igual que sucede con el elemento de datos *B*. Análogamente, la transacción T_0 tampoco lee los mismos valores de *A* y *B* que en la planificación P_0. Sin embargo, esto no ocurre en la planificación P_2, puesto que ambas transacciones perciben los mismos valores de *A* y *B* al realizar las lecturas, por lo tanto, las planificaciones P_0 y P_2 son equivalentes en cuanto a vistas.

T_0	T_1
Leer(A)	
Escribir(A)	
Leer(B)	
Escribir(B)	
	Leer(A)
	Escribir(A)
	Leer(B)
	Escribir(B)

a. Planificación P_0

T_0	T_1
	Leer(A)
	Escribir(A)
Leer(A)	
Escribir(A)	
	Leer(B)
	Escribir(B)
Leer(B)	
Escribir(B)	

b. Planificación P_1

T_0	T_1
Leer(A)	
Escribir(A)	
	Leer(A)
	Escribir(A)
Leer(B)	
Escribir(B)	
	Leer(B)
	Escribir(B)

c. Planificación P_2

Figura 3.13

3.9. SERIABILIDAD EN CUANTO A VISTAS

Una planificación es serializable en cuanto a vistas cuando es equivalente en cuanto a vistas a una planificación en serie. Tome la planificación de la figura 3.14, la cual es equivalente a la planificación secuencial $<T_0, T_1, T_2>$, puesto que T_0 lee el mismo valor de *X* en ambas planificaciones y T_2 es la última en realizar la última escritura de *X*. Por lo tanto, esta es una planificación serializable en cuanto a vistas.

T_0	T_1	T_2
Leer(X)		
	Escribir(X)	
Escribir(X)		
		Escribir(X)

Figura 3.14

3.10. PLANIFICACIONES SIN CASCADA

Aún cuando una planificación sea recuperable, puede ocurrir que sea necesario retroceder varias transacciones para recuperarse correctamente del fallo de una transacción. Esto ocurre cuando existen transacciones que leen datos que previamente han sido modificados por una transacción no comprometida.

Considere la planificación de la figura 3.15, donde T_0 modifica el elemento de datos *X*, el cual luego es leído y actualizado por la transacción T_1, y finalmente es leído por la transacción T_2. Suponga además, que la transacción T_0 no se compromete hasta que se ejecuta la operación Leer(X) correspondiente a T_2. Si T_1 falla cuando T_2 está realizando la lectura, deberá retrocederse no sólo la transacción T_0, sino también T_1 dado que la misma depende de T_0. A su vez, también será necesario retroceder T_2 ya que depende de T_1. En una situación así, el fallo de una única transacción provoca una serie de retrocesos en múltiples transacciones, denominado *retroceso en cascada*.

Nuevamente, como en el caso de las planificaciones no recuperables, tampoco es deseable la existencia de planificaciones que puedan producir retrocesos en cascada, ya que ocasionan un aumento significativo en el tiempo de procesamiento.

Para evitar este tipo de situaciones, es necesario acotar aquellas planificaciones que puedan producir retrocesos en cascada. Estas planificaciones se denominan *planificaciones sin cascada* e imponen que una transacción puede leer elementos de datos que hayan sido modificados solamente si éstas ya fueron comprometidas.

T_0	T_1	T_2
Leer(Y)		
X := Y / 2		
Escribir(X)		
	Leer(X)	
	X := X * 0,1	
	Escribir(X)	
		Leer(X)

Figura 3.15

4. PROTOCOLOS DE RECUPERACIÓN

Como se ha mencionado previamente, pueden producirse distintos tipos de fallos durante la ejecución de una transacción. Todos estos fallos tienen aparejados una potencial pérdida de integridad en la información. Es, por lo tanto, responsabilidad del *Componente de Gestión de Recuperaciones* llevar la base de datos a un estado consistente ante la ocurrencia de tales situaciones de error. Para tal fin, el Componente de Gestión de Recuperaciones debe llevar a cabo acciones concretas que permitan la correcta recuperación de la base de datos conservando las propiedades de atomicidad y durabilidad de las transacciones involucradas.

Considere el ejemplo de la figura 2.2.1.a correspondiente a la transferencia de fondos entre dos cuentas bancarias. Suponga que la transacción falla luego de realizar la operación Escribir(A). Podría considerarse como una posible solución volver a ejecutar la transacción. Claramente, esta alternativa no es una solución legítima debido a la inconsistencia resultante en la base de datos: el saldo de la cuenta *A* quedaría con un monto inferior al adecuado por una diferencia de $100.000. Otra alternativa podría consistir en no volver a ejecutar la transacción, alternativa que tampoco conserva la base de datos en un estado consistente, puesto que desde la perspectiva de la cuenta *B* nunca será percibida la transferencia, mientras que desde la cuenta *A* se observará un saldo correspondiente a una transferencia exitosa.

Puede inferirse, entonces, que ninguna de las alternativas propuestas resuelve el problema de la generación de inconsistencias en la base de datos causadas por fallos en la ejecución de transacciones. El inconveniente surge fundamentalmente debido a que la transacción no se ejecuta de manera atómica. Para alcanzar atomicidad, se presentan dos protocolos de recuperación que resuelven el problema: la recuperación basada en el registro histórico y doble paginación.

4.1. RECUPERACIÓN BASADA EN REGISTRO HISTÓRICO

El método de recuperación basada en registro histórico, también denominado *bitácora*, consiste en llevar el registro de cada una de las operaciones que modifican valores en la base de datos. Estos registros se denominan *movimientos* del registro histórico. Esta información se almacena ya que es indispensable ante la eventual ocurrencia de fallos, posibilitando la ejecución de las acciones de recuperación apropiadas. El registro histórico se almacena en disco, de forma tal que su contenido no se vea afectado más que por fallos de disco o algún evento catastrófico [Moh91] [Cru84].

Existen distintos tipos de entradas en el registro histórico. Todas ellas almacenan un identificador de transacción denotado como T_i, el cual se genera automáticamente por el sistema y su función es identificar unívocamente cada transacción. Los tipos de entradas son:

< T_i *iniciada* > : Indica el comienzo de la transacción T_i.

< T_i, *X, valor_anterior, valor_nuevo* > : Indica que la transacción T_i ha realizado una operación de escritura sobre el elemento de datos *X*, cuyo valor anterior y posterior a la modificación está dado por los campos *valor_anterior* y *valor_nuevo* respectivamente.

< T_i *comprometida* > : Indica que la transacción T_i finalizó exitosamente y establece que su efecto se debe almacenar de forma permanente en la base de datos.

< T_i *abortada* > : Indica que la transacción T_i fue abortada.

Cuando una transacción realiza una escritura, es imprescindible que cada movimiento del registro histórico correspondiente a esa escritura se almacene (en medio no volátil) primero, antes de modificar la base de datos. La modificación de la base de datos puede realizarse únicamente luego de la creación de este movimiento. Debido a esta restricción es posible deshacer cualquier cambio realizado en la base de datos, accediendo al campo *valor_anterior* de los movimientos del registro histórico y restaurando los valores de los elementos de datos por los valores correspondientes.

A fines prácticos, se asumirá que cada movimiento es escrito en almacenamiento no volátil tan pronto como se crea. También debe tenerse en cuenta que el tamaño del registro histórico puede llegar a ser extremadamente grande, dado que guarda constancia de todas las actividades de la base de datos [Sil02].

4.1.1. MODIFICACIÓN DIFERIDA DE LA BASE DE DATOS

La técnica de modificación diferida almacena todas las actualizaciones de la base de datos en el registro histórico, aplazando la ejecución de todas las operaciones escribir de una transacción hasta que la misma se compromete parcialmente, es decir, hasta que se ejecuta la última instrucción. Por lo tanto, la base de datos no ve afectado su contenido en tanto una transacción no llegue al estado parcialmente comprometida. Si el sistema falla antes de que la transacción complete su ejecución, o si la transacción aborta, la información contenida en el registro histórico simplemente se ignora, quedando la base de datos en un estado consistente.

El funcionamiento de esta técnica consiste en generar un registro <T_i *iniciada*> cuando la transacción comienza su ejecución. Cada operación Escribir(X) genera el registro de actualización correspondiente y finalmente, luego de ejecutarse la última instrucción, se crea el registro <T_i *comprometida*>. Una vez generado este registro, las escrituras diferidas tienen lugar en la base de datos. Como puede ocurrir un fallo cuando se está realizando la actualización de la misma, es necesario asegurar que todos los movimientos del registro histórico se guarden en almacenamiento no volátil. Sólo cuando puede garantizarse esta condición, la actualización real de la base de datos tiene lugar.

Es de interés destacar que esta técnica sólo requiere el nuevo valor de los elementos de datos que se modifican. Por lo tanto, puede simplificarse la estructura de los registros correspondientes a actualizaciones omitiendo el campo *valor_anterior*.

Suponga las transacciones ilustradas en la figura 4.2, donde T_0 transfiere el 10% de la cuenta *A* hacia la cuenta *B*, y T_1 realiza un depósito de $10.000 en la cuenta *C*. Suponga, además, que las transacciones se ejecutan secuencialmente y que los saldos iniciales de las cuentas antes de la ejecución son *A* = 150.000, *B* = 130.000 y *C* = 240.000.

T_0	T_1
Leer(A)	
temp := A * 0,10	
A := A - temp	
Escribir(A)	
Leer (B)	
B := B + temp	
Escribir(B)	
	Leer(C)
	C := C + 10.000
	Escribir(C)

Figura 4.2

El registro histórico correspondiente a esta planificación se ilustra en la figura 4.3:

< T_0 iniciada >
< T_0, A, 135.000 >
< T_0, B, 145.000 >
< T_0 comprometida >
< T_1 iniciada >
< T_1, C, 250.000 >
< T_1 comprometida >

Figura 4.3

Debe tenerse en cuenta que la escritura hacia la base de datos se realiza únicamente si la transacción alcanza el estado comprometida, esto significa que debe haberse alcanzado a escribir el registro *<T_i comprometida>* en el registro histórico. Por este motivo que existen diversas ejecuciones posibles como resultado de una misma planificación. Asumiendo una ejecución sin fallos, algunas de las alternativas como resultado de la ejecución de T_0 y T_1 se ilustran en las figuras 4.4.a y 4.4.b:

Registro Histórico	*Base de datos*
< T_0 iniciada >	
< T_0, A, 135.000 >	
< T_0, B, 145.000 >	
< T_0 comprometida >	
	A = 135.000
	B = 145.000
< T_1 iniciada >	
< T_1, C, 250.000 >	
< T_1 comprometida >	
	C = 250.000

a. Ordenación 1

Registro Histórico	*Base de Datos*
< T_0 iniciada >	
< T_1 iniciada >	
< T_0, A, 135.000 >	
< T_1, C, 250.000 >	
< T_1 comprometida >	
	C = 250.000
< T_0, B, 145.000 >	
< T_0 comprometida >	
	A = 135.000
	B = 145.000

b. Ordenación 2

Figura 4.4

La ordenación 1 presenta una ejecución secuencial, donde primero se ejecuta T_0 y luego T_1. La ordenación 2, en tanto, dispone de una ejecución concurrente con intercalación de operaciones de ambas transacciones. Sin embargo, puede observarse que el estado final de la base de datos es el mismo.

4.1.1.1. UTILIZACIÓN DEL REGISTRO HISTÓRICO PARA LA RECUPERACIÓN DE LA BASE DE DATOS

En caso de ocurrir un fallo durante la ejecución concurrente de transacciones, será necesario consultar el registro histórico, el cual dispone de la información necesaria para la correcta recuperación de la base de datos. Como se ha mencionado anteriormente, al procesarse una transacción, primero se guardan las modificaciones que realice la misma en el registro histórico y sólo cuando se ha generado el registro *<T_i comprometida>* es cuando las modificaciones tienen lugar en la base de datos [Zhe10].

Para llevar a cabo la recuperación se define el procedimiento *rehacer(T_i)*, el cual fija el valor de todos los elementos de datos actualizados por la transacción T_i a los nuevos valores almacenados en bitácora. Dado que podría ocurrir un nuevo fallo en el mismo momento en que se lleva a cabo la recuperación, es indispensable que el procedimiento rehacer sea idempotente, es decir, el resultado de ejecutar el proceso de recuperación sea el mismo ya sea se realice una o *N* veces. Es por este motivo que cada movimiento del registro histórico guarda el valor posterior de cada elemento de datos, y no qué operación se lleva a cabo, posibilitando el cumplimiento de esta propiedad.

Luego de ocurrir un fallo, el sistema de recuperación debe consultar el registro histórico para determinar cuáles son las transacciones que deben recuperarse. Sólo forman parte de la recuperación aquellas transacciones que poseen los movimientos *<T_i iniciada>* y *<T_i comprometida>*. Esto es así puesto que aquellas transacciones que no incluyen el

movimiento *<T_i comprometida>* son transacciones cuyas modificaciones todavía no han sido volcadas hacia la base de datos. Es por este motivo que todos los movimientos correspondientes a transacciones que no poseen tal registro, simplemente se ignoran durante el proceso de recuperación.

Suponga que se produce un fallo general antes de completarse la ejecución de las transacciones. Se analizarán tres casos posibles en los cuales la recuperación tiene lugar. Cada registro histórico ilustrado en la figura 4.5 corresponde a distintos instantes en que ocurre un fallo para la planificación de la figura 4.2.

a.	b.	c.
< T_0 iniciada >	< T_0 iniciada >	< T_0 iniciada >
< T_0, A, 135.000 >	< T_0, A, 135.000 >	< T_0, A, 135.000 >
< T_0, B, 145.000 >	< T_0, B, 145.000 >	< T_0, B, 145.000 >
	< T_0 comprometida >	< T_0 comprometida >
	< T_1 iniciada >	< T_1 iniciada >
	< T_1, C, 250.000 >	< T_1, C, 250.000 >
		< T_1 comprometida >

Figura 4.5

Suponga que ocurre un fallo luego de ejecutar Escribir(B) perteneciente a la transacción T_0. El registro histórico correspondiente se ilustra en la figura 4.5.a. Dado que no se alcanzó a escribir el registro *<T_0 comprometida>*, el proceso de recuperación solamente debe ignorar los registros históricos correspondientes a T_0.

Suponga ahora que ocurre un fallo luego de ejecutar Escribir(C) perteneciente a T_1. El registro histórico correspondiente se ilustra en la figura 4.5.b. Dado que T_0 contiene el registro *<T_0 iniciada>* y *<T_0 comprometida>*, esta transacción debe recuperarse actualizando el valor de cada elemento en la base de datos mediante el campo *valor_nuevo* del registro histórico, comenzando la restauración desde el movimiento *<T_0 iniciada>*. La transacción T_1 carece del registro *<T_1 comprometida>*, lo cual implica que el elemento de datos *C* no ha sido modificado en la base de datos. Por lo tanto, no es necesario realizar ninguna acción para recuperar esta transacción ya que el antiguo valor permanece válido.

Por último, suponga que el fallo ocurre luego de ejecutar la operación Escribir(C). Al examinar el registro histórico correspondiente en la figura 4.5.c, puede observarse que ambas transacciones contienen los registros *<T_i iniciada>* y *<T_i comprometida>*, por lo cual deben restaurarse los valores de *A*, *B* y *C* en la base de datos a los valores posteriores, mediante el campo *valor_nuevo* de cada movimiento del registro histórico.

El algoritmo de recuperación debe analizar cada movimiento del registro histórico desde atrás hacia adelante. Cada registro *<T_i comprometida>* indica que la transacción T_i debe recuperarse. En cambio, cuando una transacción contiene el registro *<T_i iniciada>* pero no posee el correspondiente registro *<T_i comprometida>*, no debe recuperarse. Recorriendo el registro de esta manera, es fácil determinar cuándo una transacción no contiene tal registro, y por lo tanto, no deben realizarse acciones de recuperación para la misma.

4.1.2. MODIFICACIÓN INMEDIATA DE LA BASE DE DATOS

La técnica de registro histórico con modificación inmediata de la base de datos realiza las modificaciones en la base de datos conforme se ejecutan las operaciones de actualización. Por lo tanto, no es necesario que una transacción finalice su ejecución para que sus cambios se vean reflejados en la base de datos. Puesto que realizar una escritura en disco por cada operación de actualización resulta muy caro operacionalmente, la implementación real de esta técnica no se realiza estrictamente de esta manera. No obstante, a fines prácticos de exponer el funcionamiento de la misma se asumirá que cada actualización se escribe inmediatamente en la base de datos.

Cada movimiento de actualización del registro histórico contiene los campos con los valores anterior y posterior. Como se presenta a continuación, el valor anterior es necesario para realizar la correcta recuperación del sistema.

Sea la planificación de la figura 4.2, con valores iniciales de las cuentas antes de la ejecución *A* = 150.000, *B* = 130.000 y *C* = 240.000. El registro histórico resultante se muestra en la figura 4.6.

< T_0 iniciada >
< T_0, A, 150.000, 135.000 >
< T_0, B, 130.000, 145.000 >
< T_0 comprometida >
< T_1 iniciada >
< T_1, C, 240.000, 250.000 >
< T_1 comprometida >

Figura 4.6

La salida de las modificaciones a la base de datos se realizan mientras que la transacción está todavía activa. Las modificaciones de datos escritas por transacciones activas se denominan *modificaciones no comprometidas*.

En la figura 4.7 se ilustran algunas posibles ejecuciones, asumiendo ausencia de fallos.

Registro Histórico	*Base de datos*
< T_0 iniciada >	
< T_0, A, 150.000, 135.000 >	
	A = 135.000
< T_0, B, 130.000, 145.000 >	
	B = 145.000
< T_0 comprometida >	
< T_1 iniciada >	
< T_1, C, 240.000, 250.000 >	
	C = 250.000
< T_1 comprometida >	

a. Ordenación 1

Registro Histórico	*Base de Datos*
< T_0 iniciada >	
< T_1 iniciada >	
< T_0, A, 150.000, 135.000 >	
< T_1, C, 240.000, 250.000 >	
	A = 135.000
	C = 250.000
< T_1 comprometida >	
< T_0, B, 130.000, 145.000 >	
	B = 145.000
< T_0 comprometida >	

b. Ordenación 2

Figura 4.7

En la ordenación 1 se dispone de una ejecución secuencial <T_0, T_1>, donde puede observarse la modificación inmediata de la base de datos luego de cada movimiento de actualización. En la ordenación 2 se ilustra una ejecución concurrente, donde también puede observarse que la modificación de la base de datos tiene lugar antes de generarse los movimientos <T_i *comprometida*>.

Puesto que las actualizaciones se realizan de manera inmediata, ante la eventual ocurrencia de un fallo no basta con ignorar aquellos movimientos del registro histórico que no hayan generado el movimiento <T_i *comprometida*>. Para aquellas transacciones en esta situación, se define el procedimiento *deshacer*(T_i), el cual recorre cada movimiento del registro histórico de aquellas transacciones no comprometidas, restaurando el valor de cada elemento de datos a su valor previo, mediante el campo *valor_anterior*. Nuevamente, las operaciones llevadas a cabo por este procedimiento deben ser idempotentes, de forma tal de garantizar una correcta recuperación incluso si un fallo se produce durante el proceso de recuperación. Además, es necesario que este procedimiento comience restaurando el valor de cada elemento de datos desde atrás hacia adelante. Para ilustrar la razón fundamental por la cual esta restricción es necesaria, considere el fragmento de registro histórico que se ilustra en la figura 4.8, correspondiente a una transacción que modifica el mismo elemento de datos en dos oportunidades distintas. Suponga inicialmente X = 1000:

< T_i iniciada >
< T_i, X, 1000, 1500 >
< T_i, X, 1500, 1800 >

Figura 4.8

Ante la ausencia del movimiento <T_i *comprometida*>, resulta necesario restaurar el valor de X al valor previo a la ejecución de T_i. Si la recuperación se realizara restaurándolo desde la última instrucción registrada en la bitácora y hacia atrás, el valor final sería X = 1500. Claramente, esta restauración no sería correcta dado que el valor inicial de X = 1000. Si en su lugar se procesa cada movimiento del registro histórico desde atrás hacia adelante, se obtiene el valor final

X = 1000. Es por este motivo que el procedimiento *deshacer(T_i)* siempre debe realizarse comenzando por el último movimiento generado en el registro histórico.

Se analizan los casos posibles en los cuales se requiere la restauración mediante los procedimientos *rehacer* y *deshacer*. Considere nuevamente la planificación de la figura 4.2 con la ejecución ordenada *<T_0, T_1>*. Suponga que el sistema cae antes de completarse las transacciones. El estado del registro histórico en cada caso se ilustra en la figura 4.9.

a.	b.	c.
< T_0 iniciada >	< T_0 iniciada >	< T_0 iniciada >
< T_0, A, 150.000, 135.000 >	< T_0, A, 150.000, 135.000 >	< T_0, A, 150.000, 135.000 >
< T_0, B, 130.000, 145.000 >	< T_0, B, 130.000, 145.000 >	< T_0, B, 130.000, 145.000 >
	< T_0 comprometida >	< T_0 comprometida >
	< T_1 iniciada >	< T_1 iniciada >
	< T_1, C, 240.000, 250.000 >	< T_1, C, 240.000, 250.000 >
		< T_1 comprometida >

Figura 4.9

En el ejemplo de la figura 4.9.a, el fallo ocurre luego de ejecutarse Escribir(B). Puesto que se encuentra el movimiento *<T_0 iniciada>* pero no *<T_0 comprometida>*, es necesario ejecutar el procedimiento *deshacer(T_0)*, restaurando los valores de *A* y *B* en el disco a 150.000 y 130.000 respectivamente.

Suponga que la caída ocurre luego de ejecutar Escribir(C). Cuando el sistema vuelve a funcionar, se deben llevar a cabo dos acciones de recuperación. En primer lugar, al recorrer el registro histórico de atrás hacia adelante se encuentra que la transacción T_0 contiene los registros *<T_0 iniciada>* y *<T_0 comprometida>*, y que la transacción T_1 sólo contiene el registro *<T_1 iniciada>*, por lo tanto, las acciones de recuperación necesarias incluyen *deshacer(T_1)* y *rehacer(T_0)*. Al finalizar la recuperación, los saldos de las cuentas son *A* = 135.000, *B* = 145.000 y *C* = 240.000. Es de importancia destacar que el procedimiento deshacer se ejecuta antes que el de rehacer. En este caso, el resultado es indistinto si se invierte el orden, pero es de suma importancia para el algoritmo de recuperación que se presentará en la próxima sección.

Por último, considere que la caída del sistema ocurre luego de escribirse en almacenamiento no volátil el movimiento *<T_1 comprometida>*. Cuando el sistema se recupera, deben rehacerse tanto T_0 como T_1, puesto que ambas incluyen en el registro histórico los movimientos *<T_0 comprometida>* y *<T_1 comprometida>*. Los saldos resultantes en la base de datos luego de las acciones de recuperación son *A* = 135.000, *B* = 145.000 y *C* = 250.000.

4.1.3. CHECKPOINTS

Para realizar una correcta recuperación debe consultarse el registro histórico para determinar cuáles son las transacciones que deben rehacerse y cuáles deshacerse. Esto significa que el mismo debe recorrerse de manera completa en busca de los movimientos *<T_i comprometida>* para determinar las acciones a llevar a cabo. Este enfoque presenta dos inconvenientes, puesto que el proceso de búsqueda consume tiempo y además se aplicarán acciones de rehacer a transacciones que ya han comprometido sus cambios en la base de datos. Aunque volver a ejecutar estas transacciones no produzca resultados erróneos, influirá directamente en el rendimiento del DBMS a la hora de llevar a cabo la recuperación del sistema. Con el objetivo de reducir esta sobrecarga, se introducen los *checkpoints* o *puntos de revisión*. El sistema se encarga de generar periódicamente checkpoints, ya sea para el esquema de modificación diferida o modificación inmediata. Cada vez que se genera un checkpoint, se debe llevar a cabo el siguiente conjunto de acciones:

1. Realizar la escritura en disco de todos los movimientos del registro histórico que se encuentren hasta ese momento en memoria principal.
2. Realizar la escritura en disco de todos los bloques de memoria intermedia que hayan sido modificados.
3. Realizar la escritura en disco de un nuevo tipo de movimiento *<checkpoint>* en el registro histórico.

Cada vez que realiza este procedimiento, no puede ejecutarse ninguna transacción que lleve a cabo operaciones de actualización.

La principal ventaja de utilizar checkpoints resulta en una mejora del rendimiento general del sistema de base de datos, puesto que se evita la reescritura de registros en disco. Conociendo aquellos movimientos del registro histórico que ya fueron actualizados en disco, es posible llevar a cabo el proceso de recuperación de manera más eficiente [Don97].

Suponga se tiene el registro histórico ilustrado en la figura 4.10. Al momento de reanudarse el sistema, luego de un fallo, es necesario realizar un proceso de recuperación. Para esto, debe examinarse el registro histórico desde atrás hacia adelante en búsqueda del primer movimiento *<checkpoint>*, correspondiente al último generado por el DBMS. Puesto que el movimiento *<checkpoint>* aparece luego del movimiento *<T_i comprometida>*, no es necesario realizar ninguna acción de rehacer para la transacción T_i, dado que todas las modificaciones realizadas por la transacción ya se han escrito en la base de datos antes de generarse el checkpoint o formado parte del mismo. Por lo tanto, no es necesario ejecutar el procedimiento rehacer sobre T_i.

Registro histórico

< T_i iniciada >
< T_i, X, 0, 2000 >
< T_i comprometida >
< checkpoint >

Figura 4.10

Esta observación permite perfeccionar los esquemas anteriores de recuperación. Cuando se produce un fallo, el esquema de recuperación examina el registro histórico para determinar la última transacción T_i que comenzó su ejecución antes de que tuviera lugar el último checkpoint. Para encontrar una transacción de este tipo, se recorre el registro histórico empezando por el final hasta encontrar el primer registro *<checkpoint>*. Luego se continúa la búsqueda hacia atrás hasta encontrar el siguiente registro *<T_i iniciada>*. Una vez que ha sido identificada la transacción T_i, sólo es necesario aplicar las operaciones rehacer y deshacer a la transacción T_i y a las transacciones T_j que comenzaron su ejecución después que T_i. De esta manera, puede ignorarse el resto del registro histórico (la parte del principio) y puede borrarse cuando se desee [Sil02].

Considere el registro histórico ilustrado en la figura 4.11. Al examinarlo, se determina cuál fue la última transacción que comenzó su ejecución antes de que tuviera lugar el último checkpoint. Para encontrar esta transacción, debe recorrerse el registro histórico comenzando por el final hasta encontrar el primer movimiento *<checkpoint>* y luego se debe continuar buscando hasta encontrar el siguiente movimiento *<T_i iniciada>*. Una vez identificada la transacción, sólo es necesario aplicar las operaciones de rehacer y deshacer a la transacción T_i y a las transacciones que comenzaron su ejecución luego de esta. En el ejemplo expuesto sólo será necesario aplicar el procedimiento *rehacer(T_i)* y *rehacer(T_k)*.

Registro histórico

< T_i iniciada >
< T_i, X, 0, 2000 >
< checkpoint >
< T_i comprometida >
< T_k iniciada >
< T_k, Y, 0, 4000 >
< T_k comprometida >

Figura 4.11

Los procedimientos de recuperación necesarios dependen del esquema de recuperación que emplea el DBMS. En caso de ser modificación diferida de la base de datos, sólo será necesario ejecutar el procedimiento rehacer, en el caso de modificación inmediata también puede ser necesario ejecutar el procedimiento deshacer. Si se utiliza esta última técnica, las operaciones de recuperación deben realizarse en el siguiente orden:

- Ejecutar el procedimiento *deshacer(T_i)* para todas las transacciones que no posean el registro *<T_i comprometida>*.
- Ejecutar el procedimiento *rehacer(T_i)* para todas las transacciones que posean el registro *<T_i comprometida>*.

4.1.3.1. CHECKPOINTS EN EJECUCIONES CONCURRENTES DE TRANSACCIONES

En la sección anterior se presentó cómo se realiza la recuperación de transacciones mediante la utilización de checkpoints en entornos monousuarios. Por este motivo, sólo se consideraba una única transacción activa (si es que había) al momento de realizar un checkpoint.

Al permitir la ejecución concurrente de transacciones, puede ocurrir que varias estén activas al momento de colocar un checkpoint en el registro histórico. Por este motivo, cada vez que se agregue un checkpoint en bitácora, deberá asociarse al mismo una lista con todas las transacciones activas en ese momento. El mismo se define como *<checkpoint L>*, donde *L* es la lista de la transacciones activas.

Nuevamente, se conserva la restricción que impide que las transacciones modifiquen los bloques de memoria intermedia y el registro histórico durante un checkpoint. Un checkpoint que permite que puedan realizarse estas modificaciones se denomina *checkpoint difuso*.

El sistema construye dos listas cuando se recupera de una caída: la *lista_deshacer*, la cual está formada por el conjunto de transacciones que deben deshacerse y la *lista_rehacer*, formada por el conjunto de transacciones que deben rehacerse. Estas listas se construyen durante el proceso de recuperación, inicialmente vacías. Luego se recorre el registro histórico hacia atrás examinando cada movimiento del mismo hasta encontrar el primer registro *<checkpoint>*. El proceso se describe a continuación:

- Por cada movimiento encontrado del estilo *<T_i comprometida>*, se agrega T_i a la *lista_rehacer*.
- Por cada movimiento encontrado de la forma *<T_i iniciada>*, si T_i no se encuentra en la *lista_rehacer*, entonces se agrega T_i a la *lista_deshacer*.

Una vez que estos movimientos del registro histórico han sido examinados, se procesa el contenido de la lista *L*. Para cada transacción T_i en *L*, si T_i no está en la *lista_rehacer*, entonces se agrega T_i a la *lista_deshacer*. Una vez finaliza la generación de ambas listas, se procede de la siguiente manera:

1. Se recorre nuevamente el registro histórico hacia atrás, comenzando en el último movimiento y se realiza una operación deshacer por cada movimiento del registro histórico que pertenezca a una transacción T_i de la *lista_deshacer*. En esta fase se

ignoran los movimientos del registro histórico concernientes a transacciones de la *lista_rehacer*. El recorrido del registro histórico termina cuando se encuentran movimientos <T_i *iniciada*> para cada transacción T_i de la *lista_deshacer*.

2. Se localiza el último movimiento <*checkpoint L*> del registro histórico. Cabe destacar que este paso puede necesitar de un recorrido del registro histórico hacia adelante si el movimiento <*checkpoint*> quedó atrás en el paso anterior.
3. Se recorre el registro histórico hacia adelante desde el último movimiento <*checkpoint L*> y se realiza una operación rehacer por cada movimiento del registro histórico que pertenezca a una transacción T_i de la *lista_rehacer*. En esta fase se ignoran los movimientos pertenecientes a transacciones de la *lista_deshacer*.

Es importante procesar el registro histórico hacia atrás en el paso 1 para garantizar que el estado resultante de la base de datos sea correcto. Luego de deshacer todas las transacciones de la *lista_deshacer*, deben rehacerse aquellas transacciones pertenecientes a la *lista_rehacer*. Nuevamente, es de suma importancia que el registro histórico sea procesado hacia adelante. Una vez completado el proceso de recuperación, se continúa con el procesamiento normal de las transacciones [Sil02].

El orden de procesamiento de las listas es importante; no hacerlo en el orden indicado podría resultar en una incorrecta recuperación de la base de datos. Considere el fragmento de registro histórico ilustrado en la figura 4.12:

< T_i, X, 300, 500 >
< T_j, X, 300, 1000 >
< T_j comprometida >

Figura 4.12

Inicialmente *X* = 300 y una transacción T_i modifica el valor de *X* a 500 y luego aborta. El rollback de esta transacción debería dejar *X* = 300. Suponga que otra transacción T_j modifica el valor de *X* a 1000, se compromete y a continuación el sistema falla. Si se procesa la *lista_rehacer* antes que la *lista_deshacer*, *X* primero tomará el valor 1000 y luego, al deshacer, tomará el valor 300, lo cual es incorrecto. El valor final de *X* debe ser 1000, garantizándose si se procesan las *lista_deshacer* y *lista_rehacer*, en ese orden.

4.3. DOBLE PAGINACIÓN

La paginación en la sombra es una técnica alternativa al esquema de recuperación basado en registro histórico. La misma consiste en dividir la base de datos en un conjunto de bloques de tamaño fijo o *páginas*, numeradas de 1 a *N*, donde *N* puede ser del orden de miles o inclusive cientos de páginas. Puesto que no es posible mantener una base de datos completa en

memoria principal, se dispone de una *tabla de páginas* con *N* entradas. Cada una de ellas contiene un puntero a una página en el disco, donde la entrada *i-ésima* apunta a la página *i-ésima* de la base de datos. El orden lógico de las páginas no tiene por qué coincidir con el orden físico en el que se encuentran almacenadas en disco, como se muestra en la figura 4.13.

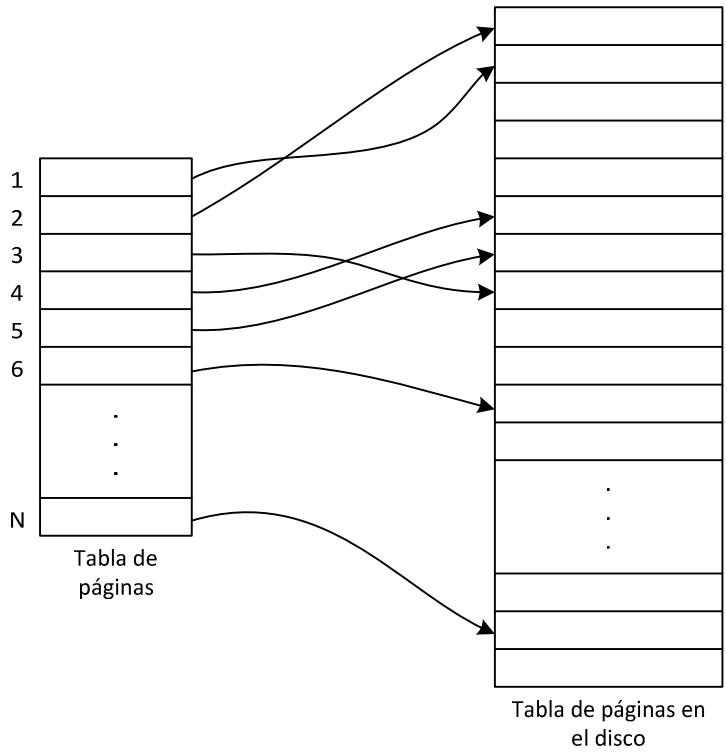


Figura 4.13

Todas las referencias a las páginas de la base de datos pasan a través de la tabla de páginas, la cual se mantiene en memoria principal. Al comienzo de una transacción, se copia el contenido de la tabla de páginas actual en una *tabla de páginas sombra*. Esta es almacenada en disco y nunca se modifica durante la ejecución de una transacción. Cuando se efectúa una operación de escritura, se crea una nueva copia de la página de la base de datos modificada, pero no se sobrescribe la copia antigua de esa página. En su lugar, se busca una nueva página en el disco que no se haya utilizado. Normalmente, el sistema de base de datos tiene acceso a una lista de páginas libres. La entrada de la tabla actual se modifica para que apunte al nuevo bloque de disco, mientras que la tabla de páginas sombra no se modifica y sigue apuntando al antiguo bloque de disco, no modificado [Kum92].

Los pasos necesarios para llevar a cabo la escritura se detallan a continuación:

1. Si la página i-ésima, la cual contiene el elemento de datos X, no se encuentra en memoria principal, se examina el disco en búsqueda de la misma y, una vez encontrada, se transfiere a memoria principal.

2. Si es la primera escritura que realiza la transacción sobre la página i-ésima, modificar la tabla actual de páginas de la siguiente manera:
    a) Encontrar una página en el disco que no se haya utilizado.
    b) Eliminar la página encontrada en el paso anterior de la lista de páginas libres.
    c) Modificar la tabla actual de páginas de forma tal que la entrada i-ésima apunte a la página encontrada en el paso anterior.
3. Asignar el valor de X de memoria principal a la página de la memoria intermedia.

Considere el ejemplo la figura 4.14, donde se debe modificar el elemento de datos *X*, el cual se encuentra en la página 3.

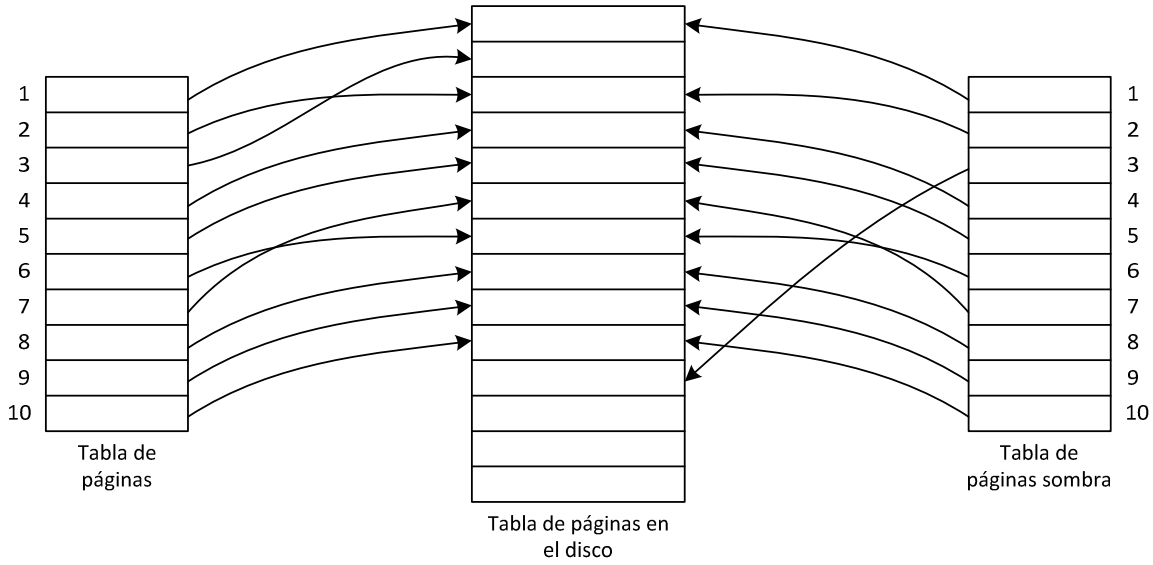


Figura 4.14

El esquema de paginación en la sombra se basa en almacenar la tabla de páginas sombra en almacenamiento no volátil, de forma tal que pueda recuperarse el estado de la base de datos antes de la ejecución de una transacción en caso de producirse un fallo o el aborto de la transacción. La tabla actual de páginas se escribe en almacenamiento no volátil cuando la transacción se compromete. Una vez llegado este punto, la tabla actual de páginas se convierte en la nueva tabla de páginas sombra y puede ejecutarse la próxima transacción. Es de suma importancia que la tabla de páginas sombra se guarde en almacenamiento no volátil, ya que es el único medio para encontrar las páginas de la base de datos.

Una ventaja de esta técnica es que las recuperaciones son automáticas, puesto que la tabla de páginas sombra apuntará a las páginas de la base de datos correspondientes al estado anterior a cualquier transacción que estuviera activa al momento de la caída. A diferencia de los esquemas basados en registro histórico, no es necesario realizar ninguna operación deshacer.

El algoritmo de recuperación se describe a continuación:

1. Asegurarse que se escriben en disco todas las páginas que se encuentran en memoria principal que hayan sido modificadas por la transacción. Cabe destacar que la escritura en disco de estas páginas no modifica el contenido de las páginas que son apuntadas por la tabla de páginas sombra.
2. Escribir en disco la tabla de páginas. No debe sobrescribirse la tabla de páginas sombra ya que puede ser necesaria para el proceso de recuperación en caso de una caída del sistema.
3. Escribir las direcciones del disco correspondientes a la tabla de páginas en el lugar del almacenamiento no volátil que contiene la dirección de la tabla de páginas sombra. Esta acción sobrescribe las direcciones de la anterior tabla de páginas sombra, por lo tanto, la tabla de páginas es ahora la tabla de páginas sombra y la transacción se compromete.

En caso de producirse una caída mientras se ejecuta una transacción basta con liberar las páginas modificadas de la base de datos y descartar la tabla de páginas actual. El estado de la base de datos previo a la ejecución de la transacción está disponible mediante la tabla de páginas sombra, por lo tanto, la recuperación se realiza simplemente haciendo que la misma sea la tabla de páginas actual. De esta manera, es fácil recuperarse de un fallo y cualquier página que se haya modificado simplemente se ignora. Si la caída ocurriera luego de completarse el paso 3, no será necesario realizar ninguna operación rehacer, puesto que se conservan los efectos de la transacción (propiedad de durabilidad) [Sil02].

Esta técnica presenta varias ventajas frente a las técnicas basadas en registro histórico. Por un lado, se elimina la sobrecarga de realizar escrituras en el registro histórico y la recuperación se realiza notablemente más rápida, puesto que no son necesarias las operaciones de rehacer ni deshacer. Sin embargo, esta técnica también presenta algunas desventajas:

- Fragmentación de datos: Las páginas actualizadas de la base de datos cambian de lugar en el disco, lo cual dificulta mucho mantener las páginas relacionadas juntas.
- Sobrecarga al comprometer transacciones: Cuando la tabla de páginas es extensa, comprometer una sola transacción implica la escritura de muchos bloques (los nuevos bloques de datos, la tabla actual de páginas y las direcciones de disco de la misma). Estas acciones incurren en un gasto adicional al tener que realizar todas estas escrituras cada vez que se compromete una transacción. Los esquemas basados en registro histórico sólo necesitan escribir los movimientos, que para transacciones típicas, caben en un solo bloque.
- Garbage collector: Las páginas que eran referenciadas por la tabla de páginas sombra y fueron actualizadas por una transacción se convierten en páginas inaccesibles. Estas páginas son consideradas como basura, puesto que ocupan espacio en el disco sin contener información útil, pero no es posible aprovecharlas, puesto que no forman parte de la lista de páginas libres. También pueden producirse páginas basura como resultado de caídas del sistema. Para evitar esto, debe implementarse un mecanismo

de garbage collector (recolección de basura) para poder recuperar estas páginas periódicamente. Este proceso requiere un sistema más complejo y sobrecargado.

- Ejecución concurrente de transacciones: El esquema de paginación en la sombra presenta más dificultades que las técnicas basadas en registro histórico para adaptarla a sistemas que permiten la ejecución concurrente de transacciones. En estos sistemas, suele ser necesaria alguna técnica basada en registro histórico, además de paginación en la sombra.

Un último aspecto a tener en cuenta es que la operación de migración entre la tabla de páginas y la tabla de páginas sombra debe implementarse como una operación atómica [Elm02].

5. HEAT

El objetivo de este capítulo es describir las características principales de HEAT (Herramienta de software para la Enseñanza de Administración de Transacciones). La motivación para su creación es complementar la transmisión de conceptos teóricos y prácticos mediante un asistente didáctico que permita comprender los lineamientos básicos en la ejecución de transacciones y cómo, mediante el uso de técnicas basadas en registro histórico o doble paginación, se asegura la integridad de la información contenida en la base de datos contra problemas generados a partir del uso cotidiano de la misma. Para lograr esto, la herramienta permite realizar la ejecución de transacciones, transmitiendo de forma clara cómo es llevada a cabo cada operación, representando el estado actual de la base de datos y de la memoria conforme se ejecutan dichas operaciones.

Para el desarrollo se utilizó Java Server Faces, un Framework basado en Java que agiliza el desarrollo de Aplicaciones Web [Web02]. Los principales componentes de esta tecnología son:

- Un conjunto de APIs para representar componentes de interfaz de usuario y manejar su estado, validación en el servidor, manejo de eventos, conversión de datos, definición de un esquema de navegación entre páginas y soporte para internacionalización y accesibilidad.
- Una biblioteca de etiquetas Java Server Pages (JSP) personalizadas para graficar componentes UI dentro de las páginas JSP.
- Manejo de eventos en el servidor.
- Administración de estados.
- Componentes Java Beans, los cuales contienen datos y funcionalidades específicas de la aplicación.

Una de las ventajas de la tecnología Java Server Faces es que ofrece una clara separación entre el comportamiento y la presentación. Además, proporciona una rica arquitectura para administrar el estado de los componentes, procesar los datos, validar entradas del usuario y manejar eventos [Web03].

Asimismo, para el desarrollo se utilizó la tecnología Ajax (Asynchronous JavaScript And XML: JavaScript asíncrono y XML), la cual permite mejorar el rendimiento de las aplicaciones Web al actualizar solamente las áreas de interés sin necesidad de recargar completamente la página, favoreciendo la obtención de mejores resultados [Web04].

5.1. CONFIGURACIÓN INICIAL

Para la simulación se propone un esquema que permite definir hasta cinco transacciones, compuestas por operaciones de lectura, actualización y escritura de datos. Asimismo, se dispone de operaciones que simulan fallo a nivel de transacción o fallo a nivel de sistema. El orden de ejecución de dichas operaciones puede ser alterado a nivel de transacción o de planificación. Admitiendo la intercalación de operaciones correspondientes a distintas transacciones, se obtienen planificaciones para entornos concurrentes.

Los valores necesarios para efectuar la simulación deben ser previamente definidos. Para ello, la herramienta prevé la inicialización de valores de configuración inicial tal cual lo indica la figura 5.1. El entorno admite la definición de cinco elementos de datos que simulan los valores de la base de datos, nombrados desde la letra A hasta la letra E, sobre los cuales se llevan a cabo las operaciones de lectura y escritura. Además, debe seleccionarse el sistema de recuperación a utilizar. Para la técnica basada en registro histórico o bitácora, las alternativas posibles son modificación inmediata o modificación diferida de la base de datos, o la técnica de doble paginación, desarrollada únicamente para entornos monousuarios.

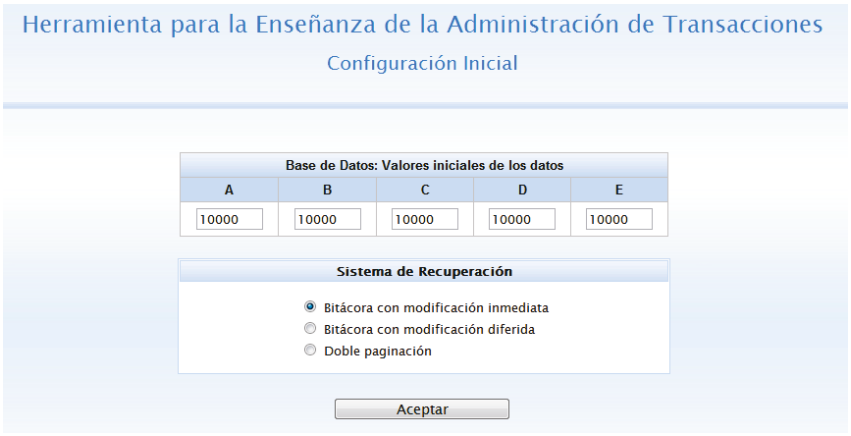


Figura 5.1

Luego de configurados los parámetros iniciales descriptos, se definen las transacciones que serán objeto de simulación. La herramienta permite definir hasta cinco transacciones, donde cada una puede acceder y modificar los cinco elementos de datos mencionados previamente. La figura 5.2 presenta la página de creación de planificaciones, dividida en las siguientes áreas:

- Título indicando el sistema de recuperación a emplear.
- Selección de tipo de operación.
- Selección de elemento de datos.
- Operación en curso, actualizada a medida que se conforma.
- Selección de operador aritmético, para operaciones de actualización binarias.

- Ingreso de constantes numéricas enteras.
- Lista de transacciones.
- Planificación.
- Botones:
    - Ejecutar planificación: Permite navegar hacia la página donde la ejecución se lleva a cabo.
    - Reiniciar: Reinicia la planificación en curso.
    - Inicio: Retorna a la página de configuración.

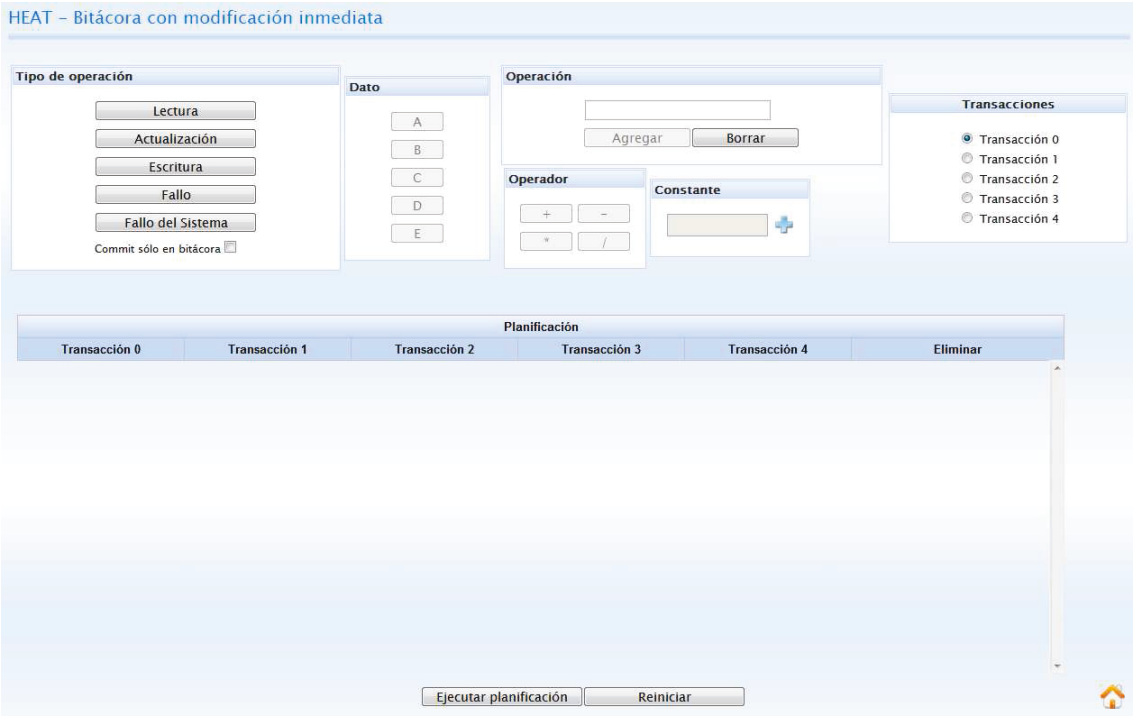


Figura 5.2

5.2. CREACIÓN DE OPERACIONES

Toda transacción ejecuta un conjunto de operaciones. Al momento de crear una operación, debe seleccionarse de la lista de transacciones activas la transacción a asociar con la operación en curso. De esta manera, las operaciones son incorporadas a la planificación a medida que son creadas.

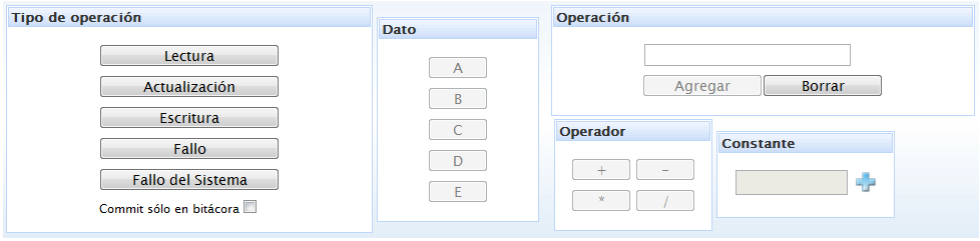
El área *tipo de operación* indica las operaciones que pueden ejecutarse. Estas operaciones son:

- Lectura: Realiza la lectura de un elemento de datos desde la base de datos hacia el área de memoria local de una transacción.
- Actualización: Permite modificar el valor de un elemento de datos previamente leído.
- Escritura: Realiza la escritura en la base de datos de un dato para una transacción.
- Fallo: Simula el fallo de una transacción.
- Fallo del Sistema: Simula el fallo a nivel de servidor de base de datos.

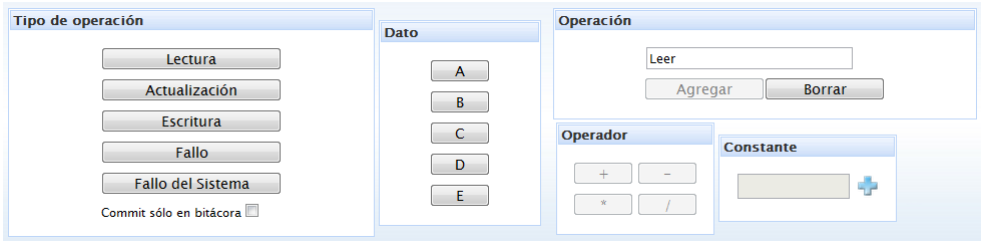
Todas las operaciones mencionadas, salvo la operación Fallo y Fallo del Sistema, necesitan definir el elemento de datos a utilizar.

5.2.1. OPERACIONES DE LECTURA Y ESCRITURA

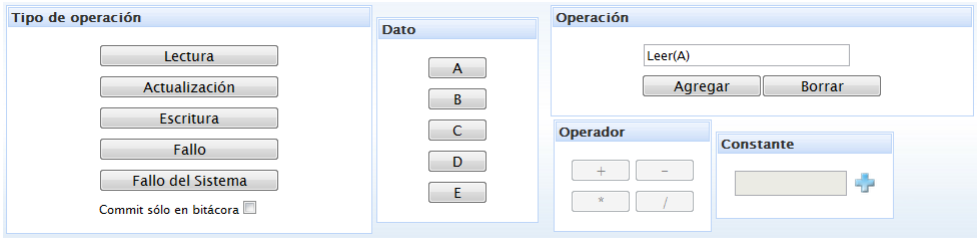
Dado que las operaciones de lectura y escritura deben efectuarse sobre un dato específico, es necesario asociar a las mismas un elemento de datos. Se dispone de un área denominada *área de datos*, que permite seleccionar el elemento de datos sobre el cual realizar la operación. La creación de una operación se ilustra paso a paso en la figura 5.3. La misma será incorporada a la columna correspondiente dependiendo de la transacción seleccionada. Este proceso se muestra en la figura 5.3.d.



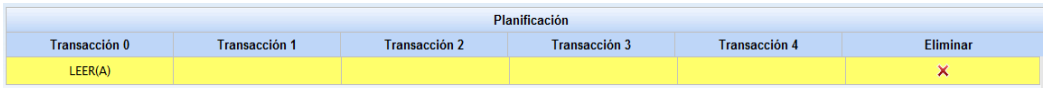
a. Antes de la selección del tipo de operación.



b. Luego de la selección del tipo de operación.



c. Luego de la selección del elemento de datos.



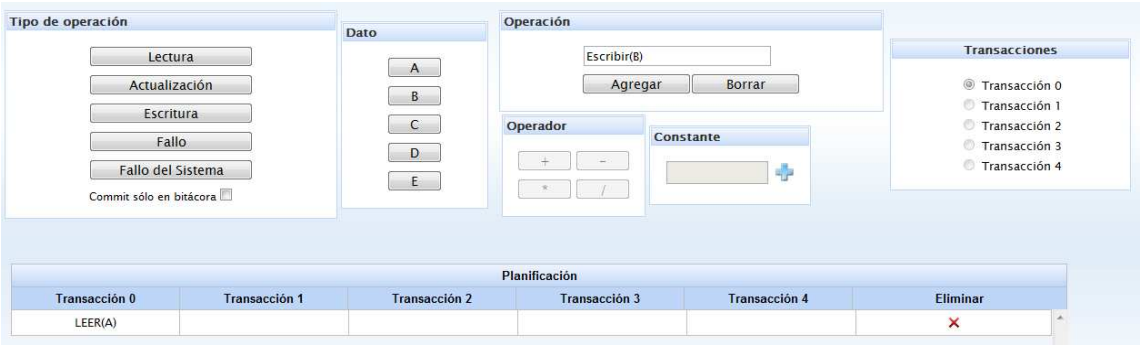
d. Planificación luego de agregar la operación.

Figura 5.3

Cabe destacar que una transacción no debería leer el mismo elemento de datos más de una vez, ya que una vez leído la transacción dispone de este valor en su área de memoria local. La validación correspondiente se ilustra en la figura 5.4, al intentar agregar nuevamente la operación Leer(A):

Figura 5.4

La creación de una operación de escritura es análoga a la de lectura. La diferencia principal radica en la imposibilidad de realizar escrituras a ciegas, es decir, escribir elementos de datos que no han sido previamente leídos. En la figura 5.5 puede observarse la validación correspondiente que impide este comportamiento.



a. Creación de una operación de escritura no válida.

La variable B no ha sido leída.

b. Informe sobre operación no válida.

Figura 5.5

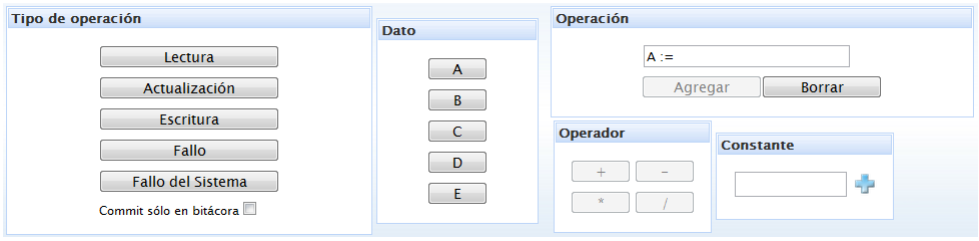
5.2.2. OPERACIÓN DE ACTUALIZACIÓN

Como precondición para crear este tipo de operación, debe tenerse en cuenta que el dato a modificar debe ser previamente leído, de manera análoga a la operación de escritura. Esta restricción ha sido establecida a fin de lograr mayor claridad en la traza de ejecución de la planificación.

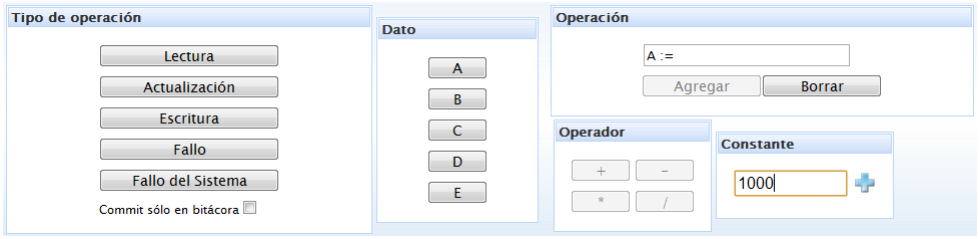
Este tipo de operación admite uno o dos operandos, tratándose en el primer caso de la operación de asignación. Los operandos pueden ser elementos de datos o bien constantes numéricas. En la siguiente sección se detalla la creación de estas operaciones.

5.2.2.1. OPERACIÓN DE ACTUALIZACIÓN UNARIA

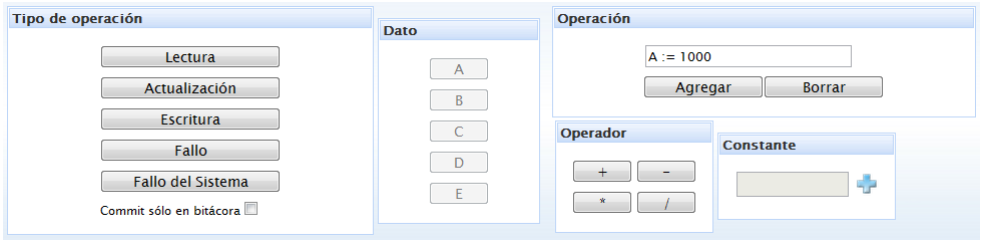
La operación de actualización con un solo operando corresponde a la operación de asignación. El operando puede ser un dato o una constante numérica. La creación de la operación de actualización unaria A := 100 se ilustra paso a paso en la figura 5.6.



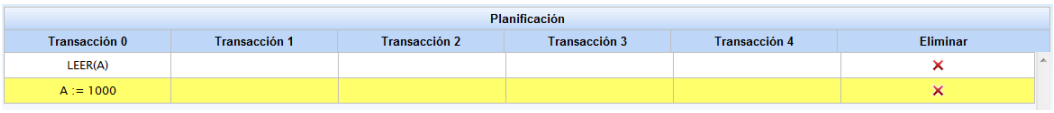
a. Selección del dato a actualizar.



b. Ingreso de una constante numérica.



c. Operación de actualización unaria.

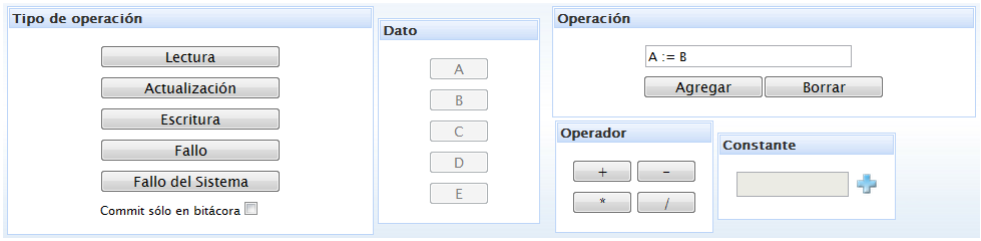


d. Planificación luego de agregar la operación.

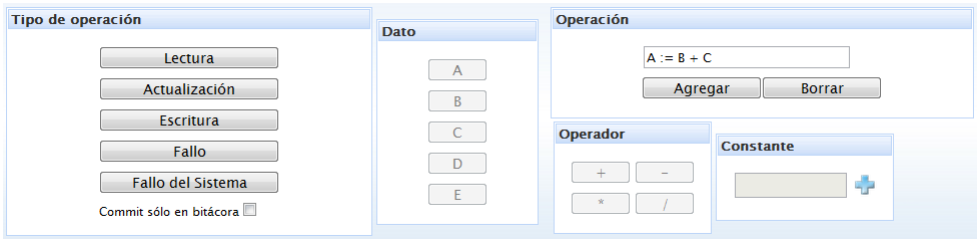
Figura 5.6

5.2.2.2. OPERACIÓN DE ACTUALIZACIÓN BINARIA

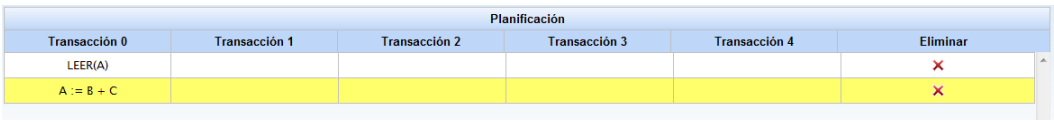
Para la creación de operaciones con dos operandos se procede de la misma manera que en el caso de un solo operando, sólo que no debe agregarse la operación a la planificación inmediatamente, sino que debe seleccionarse un operador aritmético del *área de operadores*, para luego conformar el segundo operando. Las operaciones aritméticas posibles son: suma, resta, multiplicación y división. Se ilustra la creación paso a paso de la operación A := B + C en la figura 5.7.



a. Creación del primer operando.



b. Creación del segundo operando.



c. Planificación luego de agregar la operación.

Figura 5.7

Es importante destacar que la lectura de los elementos de datos B y C es implícita, es decir, no se impone la lectura explícita de los mismos. El objetivo es obtener una mayor legibilidad y claridad en la ejecución de transacciones.

5.2.3. OPERACIÓN FALLO

La operación Fallo simula un fallo a nivel de transacción. No interesa conocer la naturaleza exacta del error, sólo interesa analizar el comportamiento de la transacción y su entorno cuando el mismo ocurre. Por este motivo, no se representan los distintos tipos de fallos existentes, sólo basta saber que una transacción no concluye de manera exitosa.

Luego de agregar la operación a la planificación, se impide la incorporación de más operaciones para la transacción. La validación se lleva a cabo deshabilitando la transacción fallida de la lista de transacciones disponibles, tal como se ilustra en la figura 5.8.

Figura 5.8

5.2.4. OPERACIÓN FALLO DEL SISTEMA

Esta operación simula un fallo a nivel de sistema, es decir, sobre el servidor. La operación de fallo total del sistema aparece al final de la planificación, siendo la última operación permitida. El fallo total del sistema refleja cómo se lleva a cabo la recuperación del sistema, en particular para los protocolos basados en registro histórico, por ser entornos multiusuario. Al ejecutar este tipo de operación, se puede examinar detalladamente cómo la utilización de una bitácora permite llevar el sistema nuevamente a un estado consistente. En el caso de doble paginación, el comportamiento de este tipo de operación y la operación fallo no difiere, por tratarse de entornos monousuario.

La opción *Commit sólo en bitácora* tiene como finalidad distinguir si el fallo del sistema se produce antes o después de generar el Commit en el registro histórico. Esta funcionalidad se describe en detalle en el próximo capítulo.

Dado que la operación Fallo del Sistema no se encuentra asociada a ningún tipo de transacción en particular, visualmente comprende todas las transacciones, como se observa en la figura 5.9.

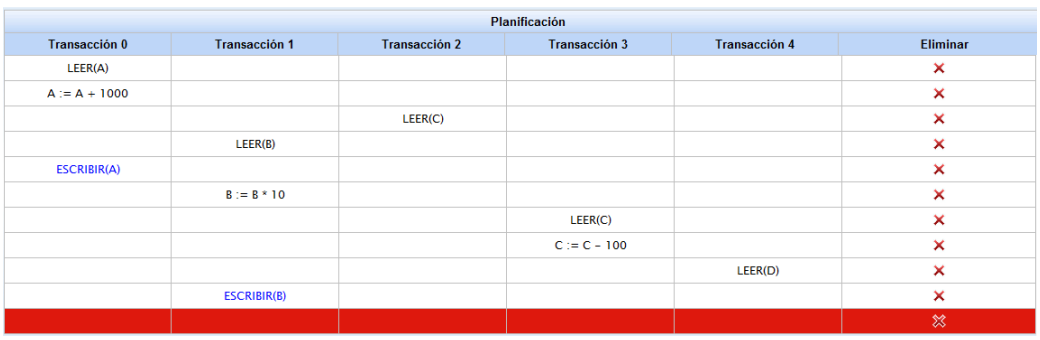


Figura 5.9

5.3. CREACIÓN DE PLANIFICACIONES

Existen dos aproximaciones para crear una planificación. La primera consiste en crear las transacciones de manera secuencial y luego reordenar las operaciones, la segunda permite alternar la selección de transacciones conforme se crean las operaciones. Para alterar el orden de ejecución la herramienta dispone dos botones que permiten alterar el orden de ejecución.

Es posible reorganizar el orden de las operaciones de una planificación. Para ello, debe contemplarse si las operaciones a reordenar pertenecen a la misma transacción. De ser así, deben respetarse las siguientes restricciones:

- Toda operación de actualización debe estar precedida por una operación de lectura.
- Toda operación de escritura debe estar precedida por una operación de lectura.
- La operación Fallo, de existir, debe ser la última operación de una transacción.
- La operación Fallo del Sistema, de existir, debe ser la última operación de la planificación.

En la figura 5.10 se ilustra paso a paso el intercambio de las primeras tres operaciones de una transacción con las tres últimas.

Transacción 0	Transacción 0	Transacción 0	Transacción 0	Transacción 0	Transacción 0
LEER(A)	LEER(B)	LEER(B)	LEER(B)	LEER(B)	LEER(B)
A := A - 1000	LEER(A)	LEER(A)	B := B + 1000	B := B + 1000	B := B + 1000
ESCRIBIR(A)	A := A - 1000	A := A - 1000	LEER(A)	LEER(A)	ESCRIBIR(B)
LEER(B)	ESCRIBIR(A)	ESCRIBIR(A)	A := A - 1000	A := A - 1000	LEER(A)
B := B + 1000	B := B + 1000	B := B + 1000	ESCRIBIR(A)	ESCRIBIR(A)	A := A - 1000
ESCRIBIR(B)	ESCRIBIR(B)	ESCRIBIR(B)	ESCRIBIR(B)	ESCRIBIR(B)	ESCRIBIR(A)

Figura 5.10

5.3.1. REORDENAMIENTO DE OPERACIONES SEGÚN EL SISTEMA DE RECUPERACIÓN

Para los sistemas basados en bitácora, el reordenamiento de operaciones está siempre permitido para operaciones pertenecientes a distintas transacciones, dando a lugar a planificaciones concurrentes. Para reordenar operaciones de una misma transacción, es necesario llevar a cabo las validaciones descriptas en la sección anterior, de forma tal de conservar su semántica.

Dado que el sistema de recuperación de doble paginación sólo admite planificaciones secuenciales, no es posible realizar intercalación de operaciones pertenecientes a distintas transacciones. No obstante, permanece válida la reordenación de operaciones a nivel de transacción.

Dada la planificación de la figura 5.11.a. Al agregar una operación a T_0, esta se inserta luego de la última operación de la transacción y no al final de la planificación (figura 5.11.b), como en el caso de los sistemas basados en bitácora. Puede observarse en la figura 5.11.c la validación correspondiente que impide la intercalación de operaciones, al intentar reordenar las operaciones Escribir(E) y Leer(C).

Transacción 0	Transacción 1
LEER(D)	
D := D + 500	
ESCRIBIR(D)	
LEER(E)	
E := E - 500	
	LEER(C)
	C := 0
	ESCRIBIR(C)

a. Planificación inicial.

Transacción 0	Transacción 1
LEER(D)	
D := D + 500	
ESCRIBIR(D)	
LEER(E)	
E := E - 500	
ESCRIBIR(E)	
	LEER(C)
	C := 0
	ESCRIBIR(C)

b. Operación insertada para doble paginación.

La planificación debe ser secuencial para doble paginación, por ser monousuario.

c. Reordenamiento no permitido.

Figura 5.11

5.4. ELIMINACIÓN DE OPERACIONES

La eliminación de operaciones es admisible para toda operación de una planificación, a excepción de la operación de lectura cuando la variable leída está sujeta a modificaciones posteriormente en la transacción. Para realizar la eliminación de una operación, debe presionarse sobre el ícono ✖. La validación correspondiente se ilustra en la figura 5.12. En este ejemplo, para eliminar la operación Leer(A), primeramente deben eliminarse las operaciones de actualización y escritura de la variable.

La eliminación de la operación Fallo implica la posibilidad de incorporar más operaciones a la transacción. Análogamente, la eliminación de la operación Fallo del Sistema admite la incorporación de operaciones a la planificación.

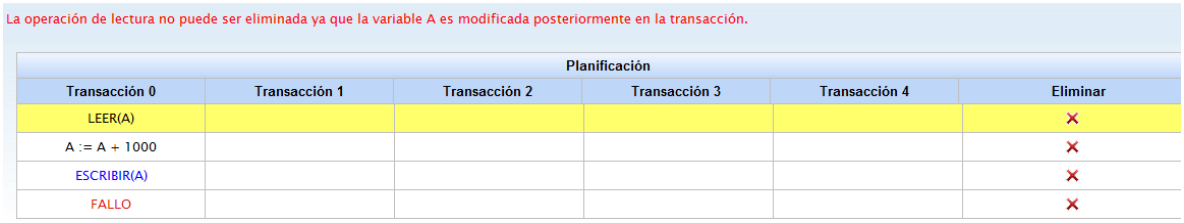


Figura 5.12

5.5. EJECUCIÓN

La ejecución de una planificación depende del sistema de recuperación seleccionado previamente en la configuración. Si bien los protocolos basados en registro histórico y doble paginación comparten la estructura general de la página, ambos métodos difieren en la forma en que la recuperación se lleva a cabo, la cual es inherente al protocolo.

Toda transacción debe comenzar con la operación *Inicio* y, en caso de ejecutarse exitosamente, debe finalizar con la operación *Commit*. La herramienta es la encargada de la inserción de estas operaciones en la planificación.

Para la técnica de doble paginación, la disposición de estas operaciones no supone complejidad alguna, puesto que las planificaciones deben ser secuenciales. La operación Inicio se inserta antes de la primera operación de cada transacción y la operación Commit se inserta al final de las mismas, ante la ausencia de fallos. En la figura 5.13.a se describe la planificación inicial, creada por el usuario, y en la figura 5.13.b se describe la planificación resultante luego de llevar a cabo dichas acciones.

Transacción 0	Transacción 1
LEER(C)	
LEER(D)	
C := C + D	
D := 0	
ESCRIBIR(C)	
ESCRIBIR(D)	
	LEER(C)

a. Planificación inicial.

Transacción 0	Transacción 1
INICIO	
LEER(C)	
LEER(D)	
C := C + D	
D := 0	
ESCRIBIR(C)	
ESCRIBIR(D)	
COMMIT	
	INICIO
	LEER(C)
	COMMIT

b. Planificación generada por HEAT.

Figura 5.13

Para los sistemas de recuperación basados en registro histórico, la disposición de las operaciones Commit no resulta tan directa. Puesto que se admite la intercalación de operaciones, determinar la correcta ubicación de cada operación Commit debe resolverse en base a las dependencias existentes en la planificación. Asimismo, la determinación de dependencias resulta fundamental ante la ocurrencia de fallos en transacciones, pues posibilita una correcta recuperación en cascada. Este concepto se desarrolla en la sección 5.5.1.1.

5.5.1. RECUPERACIÓN BASADA EN REGISTRO HISTÓRICO

El sistema de recuperación basado en registro histórico se basa en modificar el contenido de la base de datos conforme se ejecutan las operaciones de escritura (modificación inmediata), o al ejecutar la operación Commit (modificación diferida). La página donde se lleva a cabo la ejecución (figura 5.14) dispone de las siguientes áreas:

- Base de datos: Representa el contenido de la base de datos. La misma se divide en dos partes; en la parte superior se dispone del contenido inicial, es decir, el contenido antes de comenzar la ejecución, y debajo se dispone del contenido actual. Este último se actualiza conforme la base de datos. Tal duplicación tiene como objetivo cotejar el estado inicial de la misma con el resultante.
- Área de memoria local: Representa el área de memoria local para cada transacción. Cada una puede trabajar con cinco elementos de datos, los cuales inicialmente carecen de valor, representado mediante un signo de interrogación.
- Registro histórico: Dado que los sistemas de recuperación basados en registro histórico disponen de una bitácora para poder realizar una correcta recuperación del sistema ante la ocurrencia de fallos, se dispone de un sector destinado para tal fin. Este se actualiza con los movimientos correspondientes conforme se ejecuta la planificación. La estructura general de estos movimientos está dada por el sistema de recuperación.
- Planificación: La planificación a ejecutar.
- Dependencias existentes: Ante la ocurrencia de fallos, muestra las dependencias existentes entre variables pertenecientes a distintas transacciones.
- Transacciones fallidas: Comprende la lista de transacciones que deben deshacerse al momento de ejecutarse una operación Fallo.
- Botones: Controlan la ejecución de la planificación y permiten retornar a distintas áreas del entorno. Admiten las siguientes acciones:
    - Ejecución paso a paso: Ejecuta una operación a la vez.
    - Ejecución completa: Ejecuta toda la planificación.
    - Editar planificación: Retorna a la página de creación de planificaciones, posibilitando la edición de la planificación actual.

- Inicio: Retorna a la página de configuración.

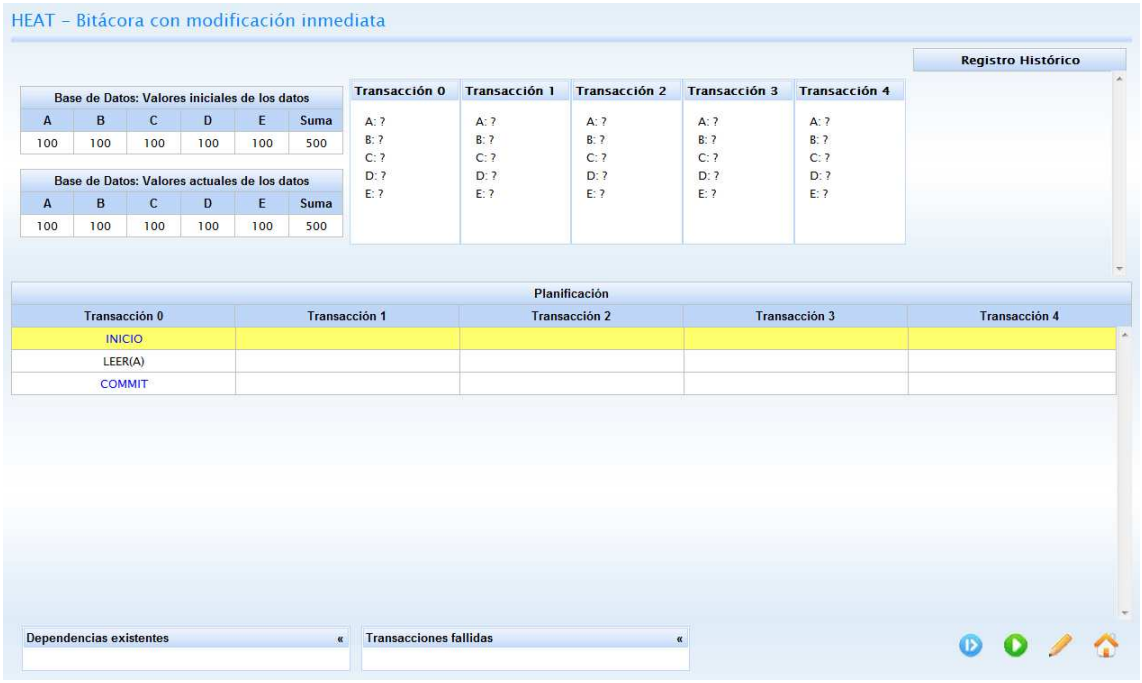


Figura 5.14

5.5.1.1. DETERMINACIÓN DE DEPENDENCIAS

Se denomina *dependencia* a la relación existente entre dos transacciones cuando una de ellas lee un elemento de datos previamente modificado y almacenado en la base de datos por otra transacción. La determinación de dependencias es necesaria dado que las mismas permiten definir, ante la ausencia de fallos, el momento preciso en que una transacción debe comprometerse.

En la sección 3.3 se ha expuesto la forma de recuperar planificaciones. Considere la planificación de la figura 5.15.a. En la misma, T_1 no puede comprometerse hasta que T_0 lo haga. La herramienta tiene como objetivo completar las planificaciones con las operaciones Inicio y Commit correspondientes. De esta manera, es posible evitar la planificación no recuperable que se ilustra en la figura 5.15.b. Este tipo de planificaciones no deben permitirse ya que generan una situación planteada anteriormente: datos no comprometidos (figura 5.15.c).

T_0	T_1
Leer(A)	
A := A + 10.000	
Escribir(A)	
	Leer(A)
Leer(B)	
B := B - 10.000	
Escribir(B)	

a. Planificación inicial.

T_0	T_1
Inicio	
Leer(A)	
A := A + 10.000	
Escribir(A)	
	Inicio
	Leer(A)
	Commit
Leer(B)	
B := B - 10.000	
Escribir(B)	
Commit	

b. Planificación no recuperable.

Transacción 0	Transacción 1
LEER(A)	
A := A + 10000	
ESCRIBIR(A)	
	LEER(A)
LEER(B)	
B := B - 10000	
ESCRIBIR(B)	

a'. Planificación inicial en HEAT.

Transacción 0	Transacción 1
INICIO	
LEER(A)	
A := A + 10000	
ESCRIBIR(A)	
	INICIO
	LEER(A)
LEER(B)	
B := B - 10000	
ESCRIBIR(B)	
COMMIT	
	COMMIT

c. Planificación generada por HEAT.

Figura 5.15

T_1 depende de manera directa de T_0. No obstante, las dependencias no siempre resultan tan evidentes. Tome la planificación inicial de la figura 5.16.a. Existe una dependencia entre T_0 y T_1 a partir de la variable A, y una dependencia entre T_1 y T_2 a partir de la variable C. Si bien T_2 no accede al elemento de datos A, se genera una dependencia de manera indirecta en la planificación, de forma tal que T_2 no puede comprometerse inmediatamente después de T_1, sino que *debe* hacerlo una vez comprometida T_0. La planificación resultante generada por la herramienta se presenta en la figura 5.16.b.

Transacción 0	Transacción 1	Transacción 2
LEER(A)		
A := A + 10000		
ESCRIBIR(A)		
	LEER(A)	
	A := A + 2000	
	ESCRIBIR(A)	
	LEER(C)	
	C := C - 2000	
	ESCRIBIR(C)	
		LEER(C)
LEER(B)		
B := B - 10000		
ESCRIBIR(B)		

a. Planificación inicial.

Transacción 0	Transacción 1	Transacción 2
INICIO		
LEER(A)		
A := A + 10000		
ESCRIBIR(A)		
	INICIO	
	LEER(A)	
	A := A + 2000	
	ESCRIBIR(A)	
	LEER(C)	
	C := C - 2000	
	ESCRIBIR(C)	
		INICIO
		LEER(C)
LEER(B)		
B := B - 10000		
ESCRIBIR(B)		
COMMIT		
	COMMIT	
		COMMIT

b. Planificación generada por HEAT.

Figura 5.16

La determinación de dependencias resulta útil ante la ocurrencia de fallos. Las mismas permiten determinar aquellas transacciones que no deben comprometerse por estar sujetas a transacciones fallidas, directa o indirectamente. De esta manera, resultan indispensables al momento de realizar la recuperación del sistema, particularmente para efectuar recuperaciones en cascada.

Sea la planificación de la figura 5.17.a. Dado que existe una dependencia entre T_0 y T_1 a partir de la variable A y, dado que T_0 falla, la transacción dependiente no puede comprometerse. De esta manera, la planificación resultante sólo difiere en las operaciones Inicio (figura 5.17.b). Las dependencias se informan en el área correspondiente (figura 5.17.c), con el objetivo de transmitir el concepto de fallo en cascada: si bien una transacción individual puede ser

correcta, esto no garantiza que la misma alcance el estado comprometida. Como se observa en el ejemplo, T_1 no alcanza tal estado pese a que el fallo no es inherente a la misma.

Transacción 0	Transacción 1
LEER(A)	
A := A + 10000	
ESCRIBIR(A)	
	LEER(A)
FALLO	

a. Planificación inicial.

Transacción 0	Transacción 1
INICIO	
LEER(A)	
A := A + 10000	
ESCRIBIR(A)	
	INICIO
	LEER(A)
FALLO	

b. Planificación generada por HEAT.

c. Informe sobre dependencias.

Figura 5.17

Si se modifica la planificación inicial, de forma tal que T_0 no alcanza un estado final correcto, se obtiene la planificación de la figura 5.18.a. Se conservan las mismas dependencias, tal como se ilustra en la figura 5.18.c:

- T_0 y T_1 generada por la variable A.
- T_1 y T_2 generada por la variable C.

Siendo T_0 una transacción fallida y, a partir de la dependencia existente entre T_0 y T_1, puede asumirse que T_1 no puede alcanzar el estado comprometida. Asimismo, T_2 depende de manera directa de T_1. Por ello y a partir de la determinación de dependencias es posible inferir que T_2 no puede comprometerse a causa del fallo en T_0, pese a la inexistencia de una relación directa entre estas transacciones. La planificación resultante se describe en la figura 5.18.b.

De esta manera, se transmiten de forma clara y directa las implicaciones que pueden producirse ante el fallo en una transacción, afectando al resto de la planificación.

Transacción 0	Transacción 1	Transacción 2
LEER(A)		
A := A + 10000		
ESCRIBIR(A)		
	LEER(A)	
	A := A + 2000	
	ESCRIBIR(A)	
	LEER(C)	
	C := C - 2000	
	ESCRIBIR(C)	
		LEER(C)
LEER(B)		
B := B - 10000		
ESCRIBIR(B)		
FALLO		

a. Planificación inicial.

Transacción 0	Transacción 1	Transacción 2
INICIO		
LEER(A)		
A := A + 10000		
ESCRIBIR(A)		
	INICIO	
	LEER(A)	
	A := A + 2000	
	ESCRIBIR(A)	
	LEER(C)	
	C := C - 2000	
	ESCRIBIR(C)	
		INICIO
		LEER(C)
LEER(B)		
B := B - 10000		
ESCRIBIR(B)		
FALLO		

b. Planificación generada por HEAT.

Dependencias existentes «

Variable A de T0 - Variable A de T1
Variable C de T1 - Variable C de T2

c. Informe sobre las dependencias.

Figura 5.18

5.5.1.2. AISLAMIENTO Y CONSISTENCIA

Es objetivo de la herramienta proporcionar los mecanismos necesarios para la generación de las planificaciones que se formulan en la teoría de la asignatura. Con la premisa de demostrar cuándo una planificación cumple con las propiedades ACID, resulta necesario elaborar contraejemplos que evidencien cuándo la propiedad de aislamiento, y en consecuencia la propiedad de consistencia, no se satisfacen. La figura 5.19 ilustra un ejemplo utilizado en la teoría de Introducción a las bases de datos que comprende uno de los ejemplos más utilizados:

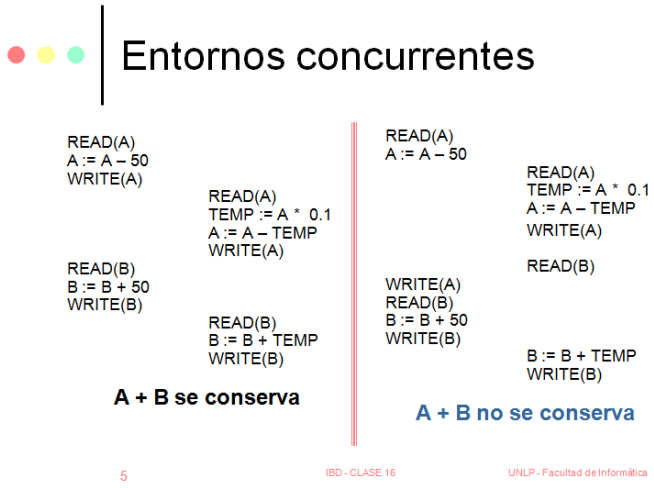


Figura 5.19

Para indicar si la propiedad de consistencia de la base de datos se conserva, el entorno proporciona la suma de los valores de la misma tanto para la base de datos inicial como para la base de datos resultante.

5.5.2. DOBLE PAGINACIÓN

El sistema de recuperación doble paginación consiste en dividir la base de datos en bloques de longitud fija, denominados *páginas*. Dado que las modificaciones se basan en localizar una página libre para escribir los nuevos valores en ella y no en modificar el dato concreto, la base de datos se plantea con un mayor número de páginas, de forma tal de exponer de manera clara el algoritmo de búsqueda y reemplazo de páginas.

Las áreas que conforman la página donde se lleva a cabo la ejecución (figura 5.20) son:

- Páginas en disco: Representa la base de datos, dividida en páginas. Las páginas inicializadas con el valor -1 indican páginas libres. Las páginas restantes contienen los valores de los datos, previamente inicializados en la configuración del entorno.
- Tabla de páginas actual: Representa la tabla de páginas actual que apunta a las nuevas páginas. Inicialmente esta tabla no es visible, dado que se crea al comienzo de la ejecución de cada transacción, es decir, al ejecutar la operación Inicio.
- Tabla de páginas sombra: Representa la tabla de páginas sombra, la cual apunta a las páginas que contienen los elementos de datos antes de comenzar la ejecución.
- Área de memoria local: Análogamente a los sistemas de recuperación basados en registro histórico, representa el área de memoria local para cada transacción. Inicialmente, los elementos de datos se representan con un signo de interrogación dado que carecen de valor.
- Planificación: La planificación secuencial a ejecutar.
- Botones: Controlan la ejecución de la planificación y permiten retornar a distintas áreas del entorno. Admiten las siguientes acciones:
    - Ejecución paso a paso: Ejecuta una operación a la vez.
    - Ejecución completa: Ejecuta toda la planificación.
    - Editar planificación: Retorna a la página de definición de planificaciones, posibilitando la edición de la planificación actual.
    - Inicio: Retorna a la página de configuración.

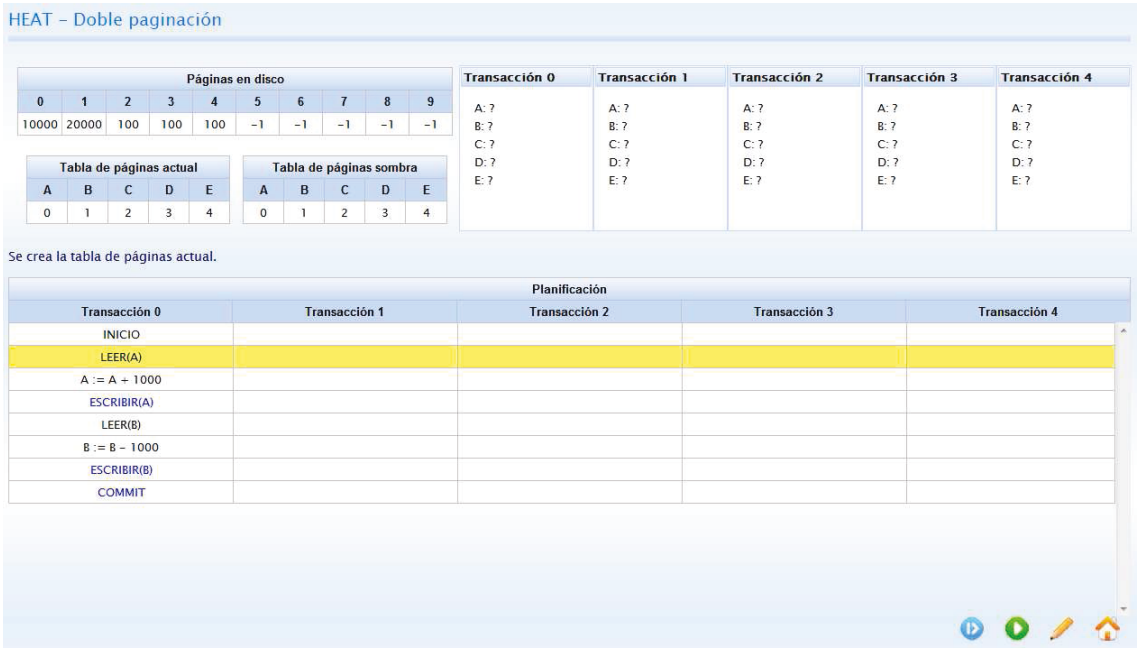


Figura 5.20

5.5.3. EJECUCIÓN PASO A PASO

Con el objetivo de lograr una clara comprensión durante la ejecución de planificaciones, se facilita la ejecución de operaciones paso a paso. De esta manera, es posible ejecutar una operación a la vez de forma tal de observar cómo se realiza la ejecución de cada operación de la planificación y cómo se lleva a cabo una correcta recuperación del sistema, ante la ocurrencia de fallos.

Los sistemas de recuperación basados en registro histórico y doble paginación comparten la forma en que se destaca visualmente la ejecución de cada operación, sólo difieren en la forma es que se realiza la recuperación, por ser inherentes al protocolo.

- Planificación: Siempre se encuentra resaltada la próxima operación a ejecutar.
- Memoria: Las operaciones de lectura y actualización modifican los valores contenidos en la memoria. De forma tal de lograr mayor claridad durante la ejecución, se destacan visualmente aquellas variables que cambian su valor en la misma conforme se ejecutan dichas operaciones.
- Base de datos: Se destaca visualmente el área de la base de datos que se actualiza al ejecutar operaciones de escritura, para los métodos de modificación inmediata y doble paginación. Para modificación diferida, la misma se actualiza al ejecutar la operación Commit.
- Área de mensajes: Se destina para proveer una mayor retroalimentación al usuario, particularmente para los procesos de recuperación.

Para los sistemas basados en bitácora, aquellas operaciones que generan movimientos en bitácora (Inicio, Escritura y Commit) se distinguen en color azul, mientras que la operación Fallo utiliza el rojo. Las operaciones ya ejecutadas se distinguen en color negro. Para la ejecución de recuperaciones se resalta el próximo movimiento a ejecutar en el registro histórico, detallando una explicación precisa en el área de mensajes conforme las mismas se llevan a cabo. Una ejecución representativa puede observarse en la figura 5.21.

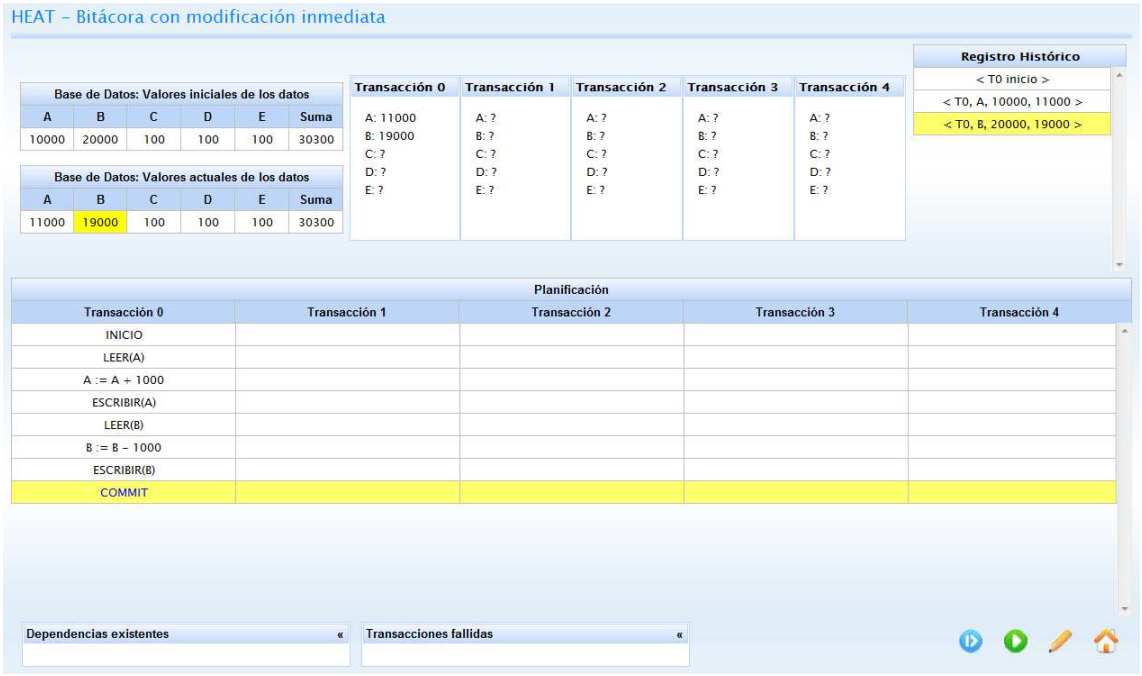


Figura 5.21

Para la técnica de doble paginación, se destacan visualmente las páginas de disco que se actualizan conforme se ejecuta una planificación, y los apuntadores de las tablas de páginas actual y sombra, como se observa en la figura 5.22.

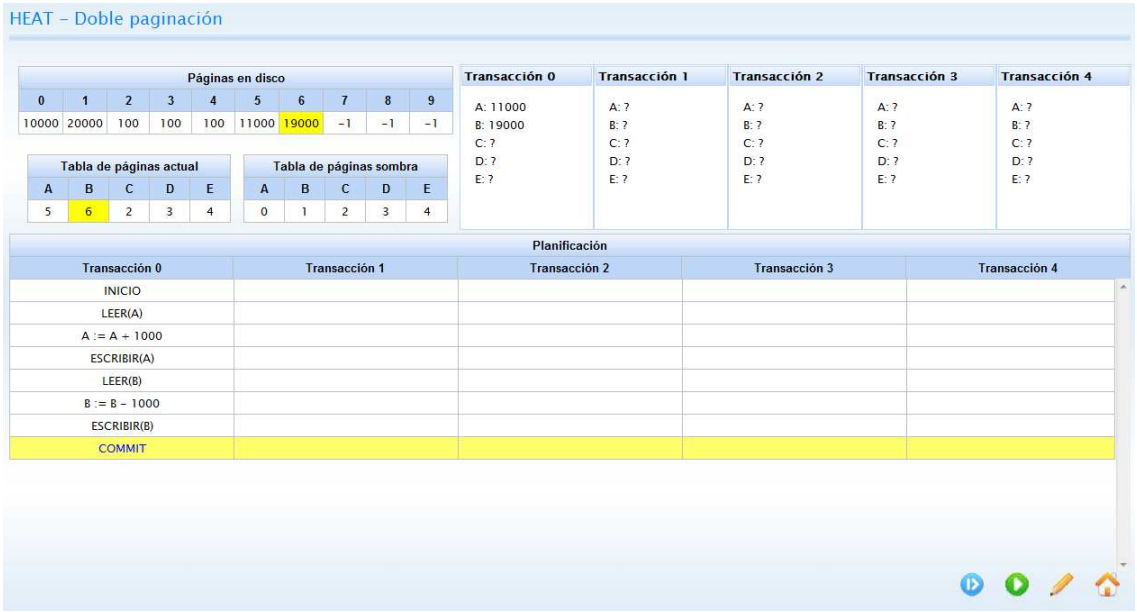


Figura 5.22

6. EJEMPLOS

En este capítulo se ejemplifican posibles usos del entorno. Para ello, se presentan ejemplos donde se muestran las dependencias, para los esquemas basados en bitácora, y donde sea necesario realizar recuperaciones de fallos tanto a nivel transaccional como a nivel de sistema. Para los casos de prueba se inicializan todos los valores de la base de datos en 100.000.

6.1. MODIFICACIÓN DIFERIDA DE LA BASE DE DATOS

Esta sección ejemplifica la utilización de la operación Fallo, la cual se encuentra asociada a una transacción particular, como ha sido descrito previamente. En la planificación de la figura 6.1.a la transacción T_0 falla. Dado que existe una dependencia entre esta transacción y T_1, la transacción T_1 no puede comprometerse. Por este motivo, tal transacción carece de la operación Commit. Asimismo, se dispone de una transacción independiente T_2, la cual sí alcanza el estado comprometida. Los informes sobre dependencias y transacciones fallidas se ilustran en la figura 6.1.b.

Como ha sido expuesto en esta tesina, el esquema de modificación diferida de la base de datos aplaza las operaciones de escritura hasta que la transacción se compromete. Al momento de ejecutar la operación Fallo, la base de datos se conserva en el estado previo a la ejecución de T_0 y T_1. De esta manera, se puede inferir que el procedimiento rehacer no tiene efecto alguno sobre la base de datos, permitiendo continuar con la ejecución normal de operaciones (figura 6.2.c). No obstante, las operaciones restantes pertenecientes a T_1 se ignoran, dado que la misma es una transacción fallida, tal como se ilustra en la figura 6.1.d.

La transacción T_2 se encuentra en ejecución al momento que T_0 falla. Puesto que no existen dependencias entre T_2 y las transacciones fallidas, esta prosigue su ejecución a pesar del fallo.

La base de datos inicial y resultante se ilustra en la figura 6.1.e. Puede observarse que las transacciones T_0 y T_1 no tienen efecto sobre la misma. El registro histórico correspondiente a la ejecución de la planificación se ilustra en la figura 6.1.f.

Transacción 0	Transacción 1	Transacción 2
LEER(A)		
A := A - 3000		
ESCRIBIR(A)		
	LEER(A)	
LEER(B)		
	A := A - 250	
	ESCRIBIR(A)	
B := B + 3000		
	LEER(D)	
ESCRIBIR(B)		
		LEER(E)
FALLO		
	D := D + 250	
	ESCRIBIR(D)	
		E := E - 100
		ESCRIBIR(E)

a. Planificación.

Dependencias existentes «
Variable A de T0 - Variable A de T1

Transacciones fallidas «
T0, T1

b. Informe sobre dependencias y transacciones fallidas.

Fallo en transacción: No es necesario rehacer movimientos ya que la base de datos no ha sido modificada.

c. Informe sobre fallo en transacción para modificación diferida.

La operación no se ejecuta ya que pertenece a una transacción fallida.

d. Informe sobre operación perteneciente a transacción fallida.

Base de Datos: Valores iniciales de los datos					
A	B	C	D	E	Suma
100000	100000	100000	100000	100000	500000

Base de Datos: Valores actuales de los datos					
A	B	C	D	E	Suma
100000	100000	100000	100000	99900	499900

e. Base de datos inicial y actual.

Registro Histórico
< T0 inicio >
< T0, A, 97000 >
< T1 inicio >
< T1, A, 99750 >
< T0, B, 103000 >
< T2 inicio >
< T0 abortada >
< T2, E, 99900 >
< T2 commit >

f. Registro histórico.

Figura 6.1

En las siguientes secciones se ejemplificará la operación Fallo del Sistema. La misma provee distintas alternativas de ejecución, acorde a la selección de la función *Commit sólo en bitácora*.

6.1.1 FALLO DEL SISTEMA

Con objetivo de ejemplificar la operación Fallo del Sistema y sus alternativas de ejecución, se dispone la planificación ilustrada en la figura 6.2.a. La última operación corresponde a la operación mencionada, la cual visualmente comprende todas las transacciones de la planificación. Al no seleccionar la opción *Commit sólo en bitácora* se indica que el fallo del sistema debe producirse inmediatamente después de la última operación ingresada. Es decir, la transacción a la cual pertenece la última operación de la planificación no debe comprometerse. La planificación generada por el entorno se ilustra en la figura 6.2.b.

El proceso de recuperación consiste únicamente en aplicar el procedimiento rehacer, por tratarse del esquema de modificación diferida. Como se ha visto en la sección previa, los cambios se graban en la base de datos una vez que se ejecuta la operación Commit. Se ilustra en la figura 6.2.c y 6.2.d el estado de la base de datos y del registro histórico al momento de producirse el fallo. Cabe destacar que las únicas modificaciones existentes en la base de datos corresponden a la transacción T_1, por ser la única que dispone del movimiento <T_i *comprometida*> en bitácora.

Para llevar a cabo la recuperación, debe recorrerse el registro histórico desde el inicio para rehacer los movimientos correspondientes. Teniendo en cuenta la importancia de proveer una continua retroalimentación al usuario, se informa el motivo por el cual un movimiento se incluye o no durante este proceso. En el ejemplo, los movimientos pertenecientes a T_0 no deben rehacerse por tratarse de una transacción fallida (figura 6.2.e), mientras que sí deben rehacerse aquellos pertenecientes a T_1 (figura 6.2.f).

El procedimiento rehacer(T_1) consiste en asignar a la variable E el valor 20.000 y a la variable D el valor 180.000; valores que ya se encuentran en la base de datos al momento de iniciar la recuperación. Puede deducirse, entonces, que el proceso de recuperación efectivamente no produce cambios. Por tal motivo, se dispone de la función *Commit sólo en bitácora*, que permite simular el fallo del sistema luego de escribir el movimiento commit en el registro histórico pero antes de almacenar los cambios en la base de datos, concepto que se desarrolla en la siguiente sección.

Transacción 0	Transacción 1
LEER(A)	
	LEER(E)
A := A - 2000	
ESCRIBIR(A)	
	E := E / 5
LEER(B)	
	ESCRIBIR(E)
	LEER(D)
B := B + 2000	
	D := D + 80000
	ESCRIBIR(D)
ESCRIBIR(B)	

a. Planificación inicial.

Transacción 0	Transacción 1
INICIO	
LEER(A)	
	INICIO
	LEER(E)
A := A - 2000	
ESCRIBIR(A)	
	E := E / 5
LEER(B)	
	ESCRIBIR(E)
	LEER(D)
B := B + 2000	
	D := D + 80000
	ESCRIBIR(D)
	COMMIT
ESCRIBIR(B)	

b. Planificación generada por HEAT.

Base de Datos: Valores iniciales de los datos					
A	B	C	D	E	Suma
100000	100000	100000	100000	100000	500000

Base de Datos: Valores actuales de los datos					
A	B	C	D	E	Suma
100000	100000	100000	180000	20000	500000

c. Base de datos inicial y actual.

Registro Histórico
< T0 inicio >
< T1 inicio >
< T0, A, 98000 >
< T1, E, 20000 >
< T1, D, 180000 >
< T1 commit >
< T0, B, 102000 >

d. Registro histórico.

Nada para rehacer. La transacción 0 no posee el movimiento Commit.

e. Procedimiento rehacer. Informe para transacción fallida.

Rehaciendo el movimiento < T1, E, 20000 >...

f. Procedimiento rehacer. Informe para operación válida.

Figura 6.2

6.1.1.1 COMMIT SÓLO EN BITÁCORA

Para describir esta funcionalidad, se toma como base el ejemplo visitado en la sección anterior, reemplazando la operación Fallo del Sistema por la que incluye la opción *Commit sólo en bitácora*. De esta manera, la planificación inicial es idéntica a la de la figura 6.2.a. La única modificación se percibe en la planificación resultante, que incorpora la operación Commit para la transacción que ejecuta la última operación de la planificación, tal como se ilustra en la figura 6.3.a.

Puede observarse que la operación Commit de T_0 se denomina *Commit bitácora*, la cual difiere de la operación Commit en lo siguiente: la primera permite simular el fallo del sistema luego de crearse el movimiento *<T_0 commit>* en el registro histórico, pero antes de almacenar las modificaciones en la base de datos, mientras que la operación Commit crea el movimiento correspondiente y luego compromete todos los cambios a disco.

El estado general del sistema previo a la ocurrencia del fallo se ilustra en la figura 6.3.b. Puede observarse que el movimiento *<T_0 commit>* efectivamente se encuentra en el registro histórico. No obstante, los nuevos valores de las variables A y B no han sido actualizados en la base de datos. Cabe destacar que estos valores dejan de estar disponibles en memoria una vez que la transacción se compromete. De esta manera, el único medio para plasmar los nuevos valores en la base de datos es mediante la utilización de la bitácora.

Una vez que se ejecuta el fallo del sistema, debe recorrerse el registro histórico desde el inicio para comenzar con el proceso de recuperación, tal cual se informa en la figura 6.3.c. El mismo consiste en rehacer únicamente aquellos movimientos pertenecientes a transacciones comprometidas (T_0 y T_1). A diferencia del caso presentado anteriormente, el procedimiento rehacer(T_0) aquí puede apreciarse, dado que al rehacer los movimientos *<T_0, A, 98.000>* y *<T_0, B, 102.000>* se actualiza la base de datos a los valores correspondientes, tal como se ilustra en la figura 6.3.d.

Finalizado el proceso de recuperación, puede observarse cómo la base de datos mantiene la consistencia, asegurando a su vez la propiedad de durabilidad.

Transacción 0	Transacción 1
INICIO	
LEER(A)	
	INICIO
	LEER(E)
A := A – 2000	
ESCRIBIR(A)	
	E := E / 5
LEER(B)	
	ESCRIBIR(E)
	LEER(D)
B := B + 2000	
	D := D + 80000
	ESCRIBIR(D)
	COMMIT
ESCRIBIR(B)	
COMMIT BITÁCORA	

a. Commit sólo en bitácora.

Base de Datos: Valores actuales de los datos					
A	B	C	D	E	Suma
100000	100000	100000	180000	20000	500000

Base de datos actual.

Transacción 0
A: 98000
B: 102000
C: ?
D: ?
E: ?

Memoria T_0.

Registro Histórico
< T0 inicio >
< T1 inicio >
< T0, A, 98000 >
< T1, E, 20000 >
< T1, D, 180000 >
< T1 commit >
< T0, B, 102000 >
< T0 commit >

Registro histórico.

b. Estado general del sistema antes de producirse el fallo.

Fallo del sistema: Por ser modificación diferida, sólo se rehacen movimientos.

c. Informe sobre fallo del sistema para modificación diferida.

Base de Datos: Valores iniciales de los datos					
A	B	C	D	E	Suma
100000	100000	100000	100000	100000	500000

Base de Datos: Valores actuales de los datos					
A	B	C	D	E	Suma
98000	102000	100000	180000	20000	500000

d. Base de datos inicial y resultante luego de la ejecución.

Figura 6.3

Cabe destacar que una transacción podría subordinar a otras transacciones, con la consecuente generación de dependencias. Tome la planificación de la figura 6.4.a. Si bien ninguna transacción falla, existen las siguientes dependencias:

- Variable A de T_1 - Variable A de T_0
- Variable A de T_1 - Variable A de T_2

- Variable B de T_2 - Variable B de T_3
- Variable D de T_3 - Variable D de T_1

El conocimiento de las mismas resulta de utilidad para comprender el orden en que deben disponerse las operaciones Commit en caso de seleccionarse la operación Fallo del Sistema con opción de Commit sólo en bitácora (figura 6.4.b). En caso contrario, la planificación resultante carece de dichas operaciones, tal como se ilustra en la figura 6.4.c. Es de interés destacar que la transacción T_4 posee la operación Commit, y no Commit bitácora, dado que es independiente de las transacciones restantes.

Planificación				
Transacción 0	Transacción 1	Transacción 2	Transacción 3	Transacción 4
	LEER(A)			
	A := A * 2			
	ESCRIBIR(A)			
		LEER(B)		
LEER(A)				
		B := B + A		
		ESCRIBIR(B)		
			LEER(B)	
			B := B + 50000	
		LEER(E)		
				LEER(C)
			LEER(D)	
		E := E - A		
			D := D - 50000	
			ESCRIBIR(B)	
		ESCRIBIR(E)		
			ESCRIBIR(D)	
	LEER(D)			

a. Planificación inicial.

Planificación				
Transacción 0	Transacción 1	Transacción 2	Transacción 3	Transacción 4
	INICIO			
	LEER(A)			
	A := A * 2			
	ESCRIBIR(A)			
		INICIO		
		LEER(B)		
INICIO				
LEER(A)				
		B := B + A		
		ESCRIBIR(B)		
			INICIO	
			LEER(B)	
			B := B + 50000	
		LEER(E)		
				INICIO
				LEER(C)
				COMMIT
			LEER(D)	
		E := E - A		
			D := D - 50000	
			ESCRIBIR(B)	
		ESCRIBIR(E)		
			ESCRIBIR(D)	
	LEER(D)			
	COMMIT BITÁCORA			
		COMMIT BITÁCORA		
COMMIT BITÁCORA				
			COMMIT BITÁCORA	

b. Fallo del sistema - Commit sólo en bitácora.

Planificación				
Transacción 0	Transacción 1	Transacción 2	Transacción 3	Transacción 4
	INICIO			
	LEER(A)			
	A := A * 2			
	ESCRIBIR(A)			
		INICIO		
		LEER(B)		
INICIO				
LEER(A)				
		B := B + A		
		ESCRIBIR(B)		
			INICIO	
			LEER(B)	
			B := B + 50000	
		LEER(E)		
				INICIO
				LEER(C)
				COMMIT
			LEER(D)	
		E := E - A		
			D := D - 50000	
			ESCRIBIR(B)	
		ESCRIBIR(E)		
			ESCRIBIR(D)	
	LEER(D)			

c. Fallo del sistema - Sin commit en bitácora.

Figura 6.4

6.2. MODIFICACIÓN INMEDIATA DE LA BASE DE DATOS

Para la siguiente simulación, se inicializan todos los valores de la base de datos en el valor 100.000. En la figura 6.5.a se dispone de la planificación a ejecutar con el sistema de recuperación basado en registro histórico con modificación inmediata. En la misma, existe una dependencia entre T_0 y T_1 generada por la variable A. Dado que T_0 falla, T_1 no puede comprometerse. Asimismo, la transacción T_2 es independiente de la ejecución de T_0 y T_1 puesto que no interviene en dependencias.

Al momento de ejecutar la operación Fallo, debe recorrerse el registro histórico hacia atrás, deshaciendo aquellos movimientos pertenecientes a transacciones fallidas. Estas incluyen las transacciones T_0 y T_1, tal como se informa en la figura 6.5.b, en conjunto con las dependencias.

La operación deshacer consiste en restaurar cada elemento de datos a su valor previo. En el ejemplo, los movimientos del registro histórico pertenecientes a T_2 se ignoran, puesto que esta transacción sí alcanza el estado comprometida. El registro histórico correspondiente a la ejecución se ilustra en la figura 6.5.d.

El estado final de la base de datos se dispone en la figura 6.5.c, donde se observa que la ejecución de las transacciones T_0 y T_1 no tuvo efecto alguno, puesto que se realizó una correcta recuperación del sistema. El único efecto observable en la misma corresponde a T_2.

Es destacable que el procedimiento rehacer no tiene comportamiento alguno para la recuperación de fallos a nivel de transacción para este esquema de recuperación, puesto que cada operación de escritura se refleja de manera inmediata en la base de datos.

Transacción 0	Transacción 1	Transacción 2
INICIO		
LEER(A)		
		INICIO
		LEER(D)
A := A + 10000		
ESCRIBIR(A)		
	INICIO	
	LEER(A)	
		LEER(C)
	A := A + 500	
		D := D - C
	ESCRIBIR(A)	
LEER(B)		
		ESCRIBIR(D)
B := B - 10000		
ESCRIBIR(B)		
FALLO		
		C := C * 2
		ESCRIBIR(C)
		COMMIT

a. Planificación generada por HEAT.

Dependencias existentes «
Variable A de T0 - Variable A de T1

Transacciones fallidas «
T0, T1

b. Informe sobre dependencias y transacciones fallidas.

Base de Datos: Valores iniciales de los datos					
A	B	C	D	E	Suma
100000	100000	100000	100000	100000	500000

Base de Datos: Valores actuales de los datos					
A	B	C	D	E	Suma
100000	100000	200000	0	100000	500000

c. Base de datos inicial y actual

Registro Histórico
< T0 inicio >
< T2 inicio >
< T0, A, 100000, 110000 >
< T1 inicio >
< T1, A, 110000, 110500 >
< T2, D, 100000, 0 >
< T0, B, 100000, 90000 >
< T0 abortada >
< T2, C, 100000, 200000 >
< T2 commit >

d. Registro histórico.

Figura 6.5

6.2.1 FALLO DEL SISTEMA

El esquema de recuperación modificación inmediata actualiza la base de datos al momento de ejecutar cada operación de escritura. Al realizar el proceso de recuperación para un fallo a nivel de sistema, debe recorrerse el registro histórico hacia atrás aplicando el procedimiento deshacer para aquellas transacciones que carecen del movimiento <T_i *commit*> y rehacer aquellas que sí lo poseen. Dado que la implementación presentada realiza las salidas hacia la base de datos por cada operación de escritura, el procedimiento rehacer no posee comportamiento alguno. De esta manera, la función *Commit sólo en bitácora* tiene como objetivo indicar que la última transacción activa debe comprometerse antes de la ocurrencia del fallo (y sus dependientes, si es que posee). En caso contrario, el fallo se produce inmediatamente después de la última operación de la planificación, impidiendo que la transacción a la cual pertenece, y sus dependientes, alcancen el estado comprometida.

Tome la planificación inicial de la figura 6.6.a. En la misma existen las siguientes dependencias:

- Variable A de T_0 - Variable A de T_1
- Variable A de T_0 - Variable A de T_2
- Variable A de T_1 - Variable A de T_2
- Variable D de T_3 - Variable D de T_4

Dado que la operación previa a la ocurrencia del fallo corresponde a T_0, interesa conocer aquellas transacciones que dependen de ella, puesto que el compromiso de estas transacciones está supeditado al compromiso de T_0.

La dependencia existente entre T_3 y T_4 no afecta a la planificación resultante, independientemente de la selección de la función *Commit sólo en bitácora*, dado que ninguna de estas transacciones se ejecuta inmediatamente antes del fallo y, a su vez, tampoco dependen de la transacción que sí lo hace.

En este caso de prueba se expone la planificación generada por el entorno sin selección de la función, la cual se ilustra en la figura 6.6.b.

Planificación				
Transacción 0	Transacción 1	Transacción 2	Transacción 3	Transacción 4
LEER(A)				
A := A - 1000				
			LEER(D)	
ESCRIBIR(A)				
	LEER(A)			
			D := D / 4	
	A := A - 2500			
			ESCRIBIR(D)	
		LEER(C)		
	ESCRIBIR(A)			
				LEER(D)
		LEER(A)		
		C := C / 2		
			LEER(E)	
		A := A + C		
			E := E + 75000	
LEER(B)				
		ESCRIBIR(A)		
			ESCRIBIR(E)	
		ESCRIBIR(C)		
B := B + 1000				
ESCRIBIR(B)				

a. Planificación inicial.

Planificación				
Transacción 0	Transacción 1	Transacción 2	Transacción 3	Transacción 4
INICIO				
LEER(A)				
A := A - 1000				
			INICIO	
			LEER(D)	
ESCRIBIR(A)				
	INICIO			
	LEER(A)			
			D := D / 4	
	A := A - 2500			
			ESCRIBIR(D)	
		INICIO		
		LEER(C)		
	ESCRIBIR(A)			
				INICIO
				LEER(D)
		LEER(A)		
		C := C / 2		
			LEER(E)	
		A := A + C		
			E := E + 75000	
LEER(B)				
		ESCRIBIR(A)		
			ESCRIBIR(E)	
			COMMIT	
				COMMIT
		ESCRIBIR(C)		
B := B + 1000				
ESCRIBIR(B)				

b. Planificación generada por HEAT.

Figura 6.6

Puede observarse que ninguna de las transacciones dependientes de T_0 posee la operación Commit. La base de datos y el registro histórico al momento del fallo se ilustran en la figura 6.7, junto con el mensaje de información que se presenta al usuario (figura 6.7.c).

Base de Datos: Valores actuales de los datos					
A	B	C	D	E	Suma
146500	101000	50000	25000	175000	497500

a. Base de datos actual.

Registro Histórico
< T0 inicio >
< T3 inicio >
< T0, A, 100000, 99000 >
< T1 inicio >
< T3, D, 100000, 25000 >
< T2 inicio >
< T1, A, 99000, 96500 >
< T4 inicio >
< T2, A, 96500, 146500 >
< T3, E, 100000, 175000 >
< T3 commit >
< T4 commit >
< T2, C, 100000, 50000 >
< T0, B, 100000, 101000 >

b. Registro histórico.

c. Informe acerca del proceso de recuperación.

Figura 6.7

Durante el proceso de recuperación, se recorre el registro histórico desde el último movimiento hasta el inicial, deshaciendo aquellos que pertenecen a las transacciones T_0, T_1 y T_2. Conforme se realiza la recuperación, el entorno informa cuál es el movimiento que se está ejecutando y se destaca visualmente el próximo movimiento a procesar en el registro histórico. Los mensajes de información más representativos correspondientes al procedimiento deshacer se ilustran en la figura 6.8.a, y los correspondientes al procedimiento rehacer en la figura 6.8.b, respetando el orden de aparición de los mismos. La finalización del proceso de recuperación se informa al usuario, tal cual se ilustra en la figura 6.8.c.

Una vez finalizado el proceso de recuperación, puede apreciarse que la base de datos satisface las propiedades de consistencia y durabilidad, al conservar la suma de los elementos de datos, tal cual se ilustra en la figura 6.8.d.

Deshaciendo el movimiento < T0, B, 100000, 101000 >...

Deshaciendo el movimiento < T2, C, 100000, 50000 >...

Nada para deshacer. La transacción 4 posee el movimiento Commit.

Nada para deshacer. La transacción 3 posee el movimiento Commit.

Deshaciendo el movimiento < T2, A, 96500, 146500 >...

Deshaciendo el movimiento < T1, A, 99000, 96500 >...

Deshaciendo el movimiento < T0, A, 100000, 99000 >...

a. Informes para el procedimiento deshacer.

Nada para rehacer. La transacción 0 no posee el movimiento Commit.

Nada para rehacer. La transacción 1 no posee el movimiento Commit.

Rehaciendo el movimiento < T3, D, 100000, 25000 >...

Nada para rehacer. La transacción 2 no posee el movimiento Commit.

Rehaciendo el movimiento < T3, E, 100000, 175000 >...

b. Informes para el procedimiento rehacer.

Fin de la recuperación.
Ejecución finalizada.

c. Informe al finalizar el proceso de recuperación.

Base de Datos: Valores iniciales de los datos					
A	B	C	D	E	Suma
100000	100000	100000	100000	100000	500000

Base de Datos: Valores actuales de los datos					
A	B	C	D	E	Suma
100000	100000	100000	25000	175000	500000

d. Base de datos inicial y resultante.

Figura 6.8

6.2.1.1 COMMIT SÓLO EN BITÁCORA

En esta sección se expone la utilización de la operación Fallo del Sistema con opción de *Commit sólo en bitácora* para el esquema de modificación inmediata. Para presentar la funcionalidad, se presenta la planificación de la figura 6.9.a, correspondiente a una reformulación de la planificación de la figura 6.6.a. Las mismas dependencias se conservan.

Esta función tiene como objetivo indicar que la última transacción activa antes del fallo debe comprometerse y, en caso de existir, también deben comprometerse todas las transacciones que de ella dependan. De esta manera, la planificación generada por el entorno se presenta en la figura 6.9.b.

Planificación				
Transacción 0	Transacción 1	Transacción 2	Transacción 3	Transacción 4
LEER(A)				
A := A - 1000				
			LEER(D)	
ESCRIBIR(A)				
	LEER(A)			
			D := D / 4	
	A := A - 2500			
			ESCRIBIR(D)	
		LEER(C)		
	ESCRIBIR(A)			
				LEER(D)
		LEER(A)		
		C := C / 2		
			LEER(E)	
		A := A + C		
			E := E + 75000	
LEER(B)				
		ESCRIBIR(A)		
			ESCRIBIR(E)	
	LEER(D)			
		ESCRIBIR(C)		
	D := D + 2500			
B := B + 1000				
	ESCRIBIR(D)			
ESCRIBIR(B)				

a. Planificación inicial.

b. Planificación generada por HEAT.

Figura 6.9

La diferencia respecto del ejemplo visto en la sección anterior radica en que las transacciones T_0, T_1 y T_2 poseen la operación Commit Bitácora. Esto quiere decir que al momento de producirse el fallo del sistema, estas transacciones disponen del movimiento *<T_i comprometida>* en el registro histórico y, por lo tanto, sólo formarán parte del procedimiento rehacer. De esta manera, el proceso de recuperación para la planificación consiste en rehacer todas las transacciones que la conforman.

La base de datos actual y el registro histórico correspondientes al momento en que ocurre el fallo del sistema se ilustran en la figura 6.10.a y 6.10.b. El proceso de recuperación consiste en rehacer las transacciones T_0, T_1, T_2, T_3 y T_4. El mismo se inicia aplicando el procedimiento deshacer a partir del último movimiento del registro histórico hasta llegar al inicio. En esta planificación, ningún movimiento se deshace puesto que todas las transacciones se encuentran comprometidas. Luego, se aplica el procedimiento rehacer desde el primer movimiento del registro histórico hasta el final.

Dado que no se deshace ningún movimiento, todos los mensajes de información correspondientes a este caso de prueba son idénticos (figura 6.10.c). Los mensajes de información más representativos correspondientes al procedimiento rehacer se ilustran en la figura 6.10.d, respetando el orden de aparición de los mismos.

Una vez finalizado el proceso de recuperación, la base de datos satisface las propiedades de consistencia y durabilidad, al conservar la suma de los elementos de datos, tal cual se ilustra en la figura 6.10.e.

Base de Datos: Valores actuales de los datos					
A	B	C	D	E	Suma
146500	101000	50000	27500	175000	500000

a. Base de datos actual.

Registro Histórico
< T0 inicio >
< T3 inicio >
< T0, A, 100000, 99000 >
< T1 inicio >
< T3, D, 100000, 25000 >
< T2 inicio >
< T1, A, 99000, 96500 >
< T4 inicio >
< T2, A, 96500, 146500 >
< T3, E, 100000, 175000 >
< T3 commit >
< T4 commit >
< T2, C, 100000, 50000 >
< T1, D, 25000, 27500 >
< T0, B, 100000, 101000 >
< T0 commit >
< T1 commit >
< T2 commit >

b. Registro histórico.

Nada para deshacer.

c. Informe para el procedimiento deshacer.

Rehaciendo el movimiento < T0, A, 100000, 99000 >...

Rehaciendo el movimiento < T3, D, 100000, 25000 >...

Rehaciendo el movimiento < T1, A, 99000, 96500 >...

Rehaciendo el movimiento < T2, A, 96500, 146500 >...

Rehaciendo el movimiento < T3, E, 100000, 175000 >...

Rehaciendo el movimiento < T2, C, 100000, 50000 >...

Rehaciendo el movimiento < T1, D, 25000, 27500 >...

Rehaciendo el movimiento < T0, B, 100000, 101000 >...

d. Informes para el procedimiento rehacer.

Base de Datos: Valores iniciales de los datos					
A	B	C	D	E	Suma
100000	100000	100000	100000	100000	500000

Base de Datos: Valores actuales de los datos					
A	B	C	D	E	Suma
146500	101000	50000	27500	175000	500000

e. Base de datos inicial y resultante.

Figura 6.10

6.3. DOBLE PAGINACIÓN

En esta sección se expone la utilización del sistema de recuperación basado en doble paginación. Se presentan distintas planificaciones donde se enseña su funcionamiento para condiciones normales de ejecución y para ocurrencia de fallos. Dado que este esquema se implementa para entornos monousuarios, las operaciones Fallo y Fallo del Sistema no difieren en su comportamiento. Por ende, el proceso de recuperación es análogo para ambas.

Se dispone la planificación inicial ilustrada en la figura 6.11.a y la correspondiente planificación generada por el entorno en la figura 6.11.b. Es de interés recordar que las planificaciones deben, necesariamente, ser secuenciales. La herramienta impide la generación de planificaciones concurrentes para doble paginación.

Planificación	
Transacción 0	**Transacción 1**
LEER(C)	
C := C + 25000	
ESCRIBIR(C)	
LEER(D)	
D := D – 25000	
ESCRIBIR(D)	
	LEER(C)
	C := C – 40000
	LEER(E)
	E := E + 40000
	ESCRIBIR(C)
	ESCRIBIR(E)

a. Planificación inicial.

Planificación	
Transacción 0	**Transacción 1**
INICIO	
LEER(C)	
C := C + 25000	
ESCRIBIR(C)	
LEER(D)	
D := D – 25000	
ESCRIBIR(D)	
COMMIT	
	INICIO
	LEER(C)
	C := C – 40000
	LEER(E)
	E := E + 40000
	ESCRIBIR(C)
	ESCRIBIR(E)
	COMMIT

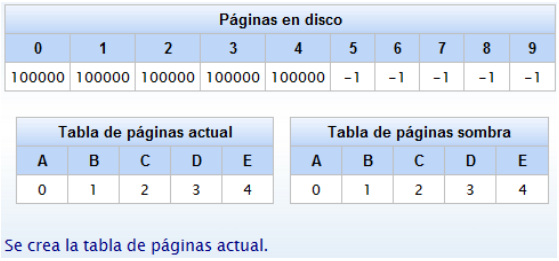
b. Planificación generada por HEAT.

Figura 6.11

Para la simulación, se inicializan todos los valores de la base de datos en el valor 100.000. Al ejecutar la operación Inicio de cada transacción, se crea la tabla de páginas actual. Inicialmente, esta es idéntica a la tabla de páginas sombra, tal cual se ilustra en la figura 6.12.a. La transacción T_0 lee y modifica la variable C. Al realizar la escritura, se debe buscar una nueva página libre (página 5) y actualizar la referencia a la misma (figura 6.12.b). Asimismo, para la escritura de la variable D se debe buscar una nueva página, la 6 en este caso, y actualizar la referencia. Una vez que la transacción se compromete, la tabla de páginas actual pasa a ser la nueva tabla de páginas sombra y se señalan como disponibles las páginas que ya no se utilizan (figura 6.12.d). Luego, la tabla de páginas actual se libera.

Cuando T_1 precisa una nueva página libre para realizar la escritura de C, el algoritmo de búsqueda y reemplazo de páginas devuelve la primera página libre disponible, en este caso, la página 2. La tabla de páginas actual actualiza la nueva referencia a C. De la misma manera,

para la escritura de E, el algoritmo devuelve la página de disco número 3 y actualiza la referencia. Nuevamente, la transacción se compromete y la tabla actual de páginas pasa a ser la nueva tabla de páginas sombra. Las páginas en disco luego de la ejecución se ilustran en la figura 6.12.e.



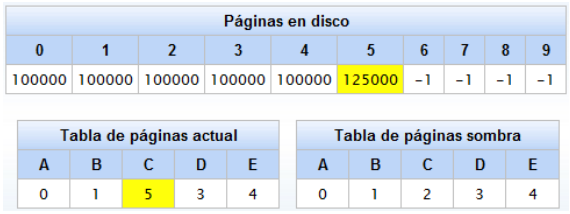
Páginas en disco									
0	1	2	3	4	5	6	7	8	9
100000	100000	100000	100000	100000	-1	-1	-1	-1	-1

Tabla de páginas actual				
A	B	C	D	E
0	1	2	3	4

Tabla de páginas sombra				
A	B	C	D	E
0	1	2	3	4

Se crea la tabla de páginas actual.

a. Luego de ejecutar la operación Inicio.

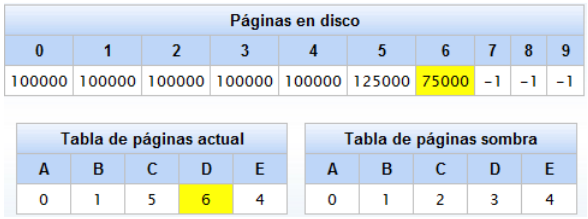


Páginas en disco									
0	1	2	3	4	5	6	7	8	9
100000	100000	100000	100000	100000	125000	-1	-1	-1	-1

Tabla de páginas actual				
A	B	C	D	E
0	1	5	3	4

Tabla de páginas sombra				
A	B	C	D	E
0	1	2	3	4

b. Escritura del dato C.

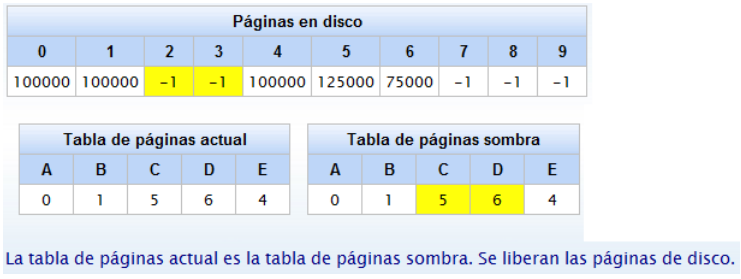


Páginas en disco									
0	1	2	3	4	5	6	7	8	9
100000	100000	100000	100000	100000	125000	75000	-1	-1	-1

Tabla de páginas actual				
A	B	C	D	E
0	1	5	6	4

Tabla de páginas sombra				
A	B	C	D	E
0	1	2	3	4

c. Escritura del dato D.



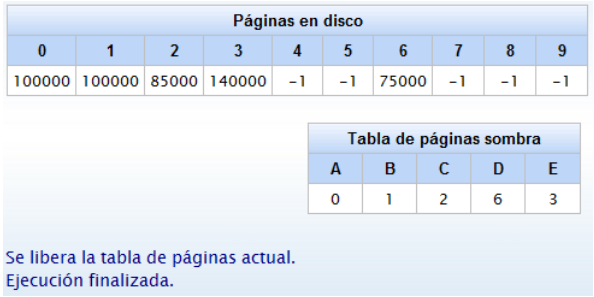
Páginas en disco									
0	1	2	3	4	5	6	7	8	9
100000	100000	-1	-1	100000	125000	75000	-1	-1	-1

Tabla de páginas actual				
A	B	C	D	E
0	1	5	6	4

Tabla de páginas sombra				
A	B	C	D	E
0	1	5	6	4

La tabla de páginas actual es la tabla de páginas sombra. Se liberan las páginas de disco.

d. Liberación de páginas y actualización de la tabla sombra.



Páginas en disco									
0	1	2	3	4	5	6	7	8	9
100000	100000	85000	140000	-1	-1	75000	-1	-1	-1

Tabla de páginas sombra				
A	B	C	D	E
0	1	2	6	3

Se libera la tabla de páginas actual.
Ejecución finalizada.

e. Fin de la ejecución.

Figura 6.12

6.3.1 FALLO

Se presenta aquí la utilización de la operación Fallo. Por tratarse de un entorno monousuario, el proceso de recuperación no difiere del llevado a cabo para la operación Fallo del Sistema. De esta manera, resulta trivial la selección de una u otra operación para la exposición del tema. La única diferencia estriba en la operación Fallo, que por producirse a nivel transaccional, admite la incorporación de más operaciones a la planificación, mientras que la operación Fallo del Sistema debe, necesariamente, ser la última operación en la misma.

Tome la planificación de la figura 6.13.a, correspondiente a la planificación de la sección previa con la incorporación de la operación Fallo. Dado que T_0 falla, esta carece de la operación Commit, tal cual se ilustra en la figura 6.13.b.

Planificación	
Transacción 0	Transacción 1
LEER(C)	
C := C + 25000	
ESCRIBIR(C)	
LEER(D)	
D := D - 25000	
ESCRIBIR(D)	
FALLO	
	LEER(C)
	C := C - 40000
	LEER(E)
	E := E + 40000
	ESCRIBIR(C)
	ESCRIBIR(E)

a. Planificación inicial.

Planificación	
Transacción 0	Transacción 1
INICIO	
LEER(C)	
C := C + 25000	
ESCRIBIR(C)	
LEER(D)	
D := D - 25000	
ESCRIBIR(D)	
FALLO	
	INICIO
	LEER(C)
	C := C - 40000
	LEER(E)
	E := E + 40000
	ESCRIBIR(C)
	ESCRIBIR(E)
	COMMIT

b. Planificación generada por HEAT.

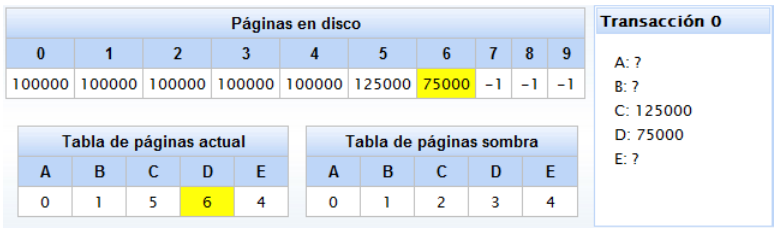
Figura 6.13

Hasta el momento previo a la ocurrencia del fallo, la ejecución de la transacción T_0 es idéntica a la del ejemplo anterior. El estado general del sistema se ilustra en la figura 6.14.a. Al producirse el fallo, la tabla actual de páginas simplemente se desecha y las páginas utilizadas por la transacción fallida se marcan como disponibles (figura 6.14.b).

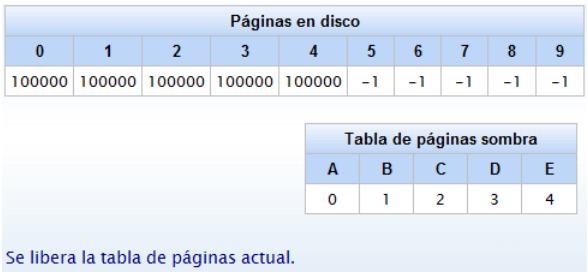
Finalizado el proceso de recuperación, se procede con la ejecución de la siguiente transacción. La ejecución de la operación Commit se ha dividido en dos pasos para lograr una clara presentación de su comportamiento: primero, se actualiza la tabla de páginas sombra y las páginas en disco, y en una segunda instancia, se libera la tabla actual de páginas.

El estado final del sistema una vez finalizada la ejecución se ilustra en la figura 6.14.c. Puede observarse que la base de datos satisface las propiedades ACID. Los valores finales

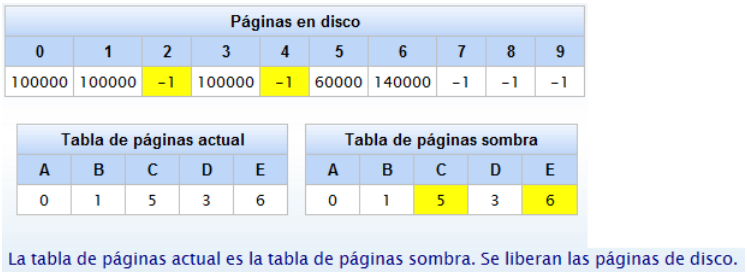
corresponden a la ejecución exitosa de T_1, mientras que la ejecución de T_0 no tiene efecto alguno, respetando así la propiedad de atomicidad.



a. Páginas en disco y memoria de T_0 antes del fallo.



b. Luego del fallo de T_0.



c. Actualización de la tabla sombra y páginas en disco.

Figura 6.14

7. CONCLUSIONES

HEAT es un asistente didáctico cuyo principal objetivo es la transmisión de conceptos teóricos y prácticos en la enseñanza de transacciones. Mediante su utilización, el alumno posee una herramienta que permite comprender el funcionamiento de transacciones y cómo, mediante el uso de técnicas basadas en bitácora o doble paginación, se asegura la integridad de la información contenida en la base de datos contra problemas generados a partir del uso cotidiano de la misma.

Si bien el tema de transacciones, seguridad e integridad de datos es un tema teórico para la asignatura Introducción a las Bases de Datos, resulta de gran interés contar con una herramienta que permite mostrar de manera simulada el uso de transacciones para garantizar la consistencia de los datos.

En la primera parte de este trabajo se realizó un extenso análisis sobre los conceptos ligados a transacciones, transacciones concurrentes y planificaciones, y se evaluaron los distintos sistemas de recuperación existentes para transmitir, a alto nivel, el conjunto de acciones que lleva a cabo un sistema de bases de datos para recuperarse correctamente de los fallos y garantizar las propiedades ACID.

La herramienta admite la generación de un sinnúmero de casos de prueba, permitiendo además la configuración dinámica de los mismos. De esta manera, el usuario puede simular distintas ejecuciones y comparar los resultados obtenidos, favoreciendo una correcta transmisión de los conceptos teóricos que, en conjunto, garantizan la integridad de la información en una base de datos.

La herramienta se adapta a los requerimientos iniciales de la cátedra y cumple los objetivos propuestos para esta tesina, permitiendo que el alumno pueda generar sus propios casos de prueba y observar cómo se lleva a cabo la ejecución de transacciones y la recuperación ante la ocurrencia de fallos tanto a nivel de transacción como a nivel de sistema, transmitiendo los conceptos de manera sencilla y didáctica.

8. TRABAJO FUTURO

La línea de trabajo actual carece de la eficiencia que podría obtenerse mediante la utilización de checkpoints. El proceso de recuperación siempre recorre el registro histórico íntegramente para restaurar la base de datos a los valores correspondientes. Aunque volver a ejecutar las transacciones que ya han escrito sus valores en la base de datos no produce resultados erróneos, sí influye en la eficiencia, ya que el proceso de recuperación toma más tiempo del necesario.

Asimismo, ante la ocurrencia de fallos para los esquemas basados en bitácora, el procedimiento rehacer en ocasiones carece de comportamiento, dado que la implementación actual simula los cambios directamente sobre la base de datos. Una alternativa es administrar un buffer de almacenamiento intermedio, impidiendo así que las modificaciones se realicen directamente sobre la base de datos.

Otra funcionalidad que podría implementarse es la inspección del estado general del sistema al momento en que se ejecutó una operación dada. Esto incluye los valores contenidos en la memoria de cada transacción, los valores de la base de datos, el registro histórico para los esquemas basados en bitácora, y las tablas de páginas para doble paginación.

9. REFERENCIAS

[Dat01] Introducción a los Sistemas de Bases de Datos. C. J. Date. Pearson Educación. 2001. 7° Ed.

[Kro10] Procesamiento de Bases de Datos. David E. Kroenke. Prentice Hall. 2010. 11° Ed.

[Sil02] Fundamentos de Bases de Datos. Abraham Silberschatz, Henry F. Korth, S. Sudarshan. 2002. 4° Ed.

[Elm02] Fundamentos de Sistemas de Bases de Datos. Elmasri, Navathe. Pearson Addison Wesley. 2002.

[Rob02] Sistemas de Bases de Datos. Peter Rob, Carlos Coronel. Thomson. 2002.

[Han97] Administración de Bases de Datos. Hansen Hansen. Prentice Hall. 1997.

[Sil99] Sistemas Operativos. Abraham Silberschatz. 1999.

[Ber11] Introducción a las Bases de Datos: fundamentos y diseño. Rodolfo Bertone, Pablo Thomas. 2011. Prentice Hall. 1° Ed.

[Uns04] Universidad Nacional del Sur. Departamento de Ciencias e Ingeniería de la Computación. Elementos de Bases de Datos - 2do. Cuatrimestre de 2004

[Kin90] Overview of Disaster Recovery for Transaction Processing Systems. Richard P. King, Nagui Halim. 10th International Conference on Distributed Computing Systems. 1990.

[Moh91] ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method. C. Mohan, Hamid Pirahesh. 7th International Conference on Data Engineering. 1991.

[Kum92] Performance Measurement of Main Memory Database Recovery Algorithms Based on Update-in-Place and Shadow Approaches. Vijay Kumar, Albert Burger. IEEE Transactions on Knowledge and Data Engineering. 1992.

[Don97] Checkpointing Schemes for Fast Restart in Main Memory. Database System. Dongho Lee, Haengrae Cho. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing. '10 Years PACRIM 1987-1997 - Networking the Pacific Rim'. 1997.

[Yad03] Analysis of Cooperation Semantics for Transaction Processing with Full and Partial Aborts in Active Database. D. S. Yadav, Rajeev Agrawal, R. C. Saraswat. Proceedings of the 2003 Joint Conference of the Fourth International Conference on Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. 2003.

[Ham06] Checking Integrity Constraints - How it Differs in Centralized, Distributed and Parallel Databases. Hamidah Ibrahim. 17th International Workshop on Database and Expert Systems Applications. DEXA '06. 2006.

[Hof07] Recovering from Database Recovery: Case Studies and the Lessons They Teach. IAW '07. IEEE SMC. Information Assurance and Security Workshop. 2007.

[Zhe10] The Advanced Data Recovery Technology Based On the Log Recovery. Zheng Zhi-gao, Wang Yu-ting. 2010 International Conference on Internet Technology and Applications. 2010.

[Cru84] Data recovery in IBM Database 2. R.A. Crus. IBM General Products Division, Santa Teresa Laboratory. 1984.

[Wie10] Handbook of nanoscale optics and electronics, p. 257. Wiederrecht Gary Phillip. Academic Press. 2010.

[Web01] http://www.fdi.ucm.es/ Facultad de Informática. Universidad Complutense. http://www.fdi.ucm.es/profesor/milanjm/bdsi0304/Tema_06.pdf

[Web02] http://www.javaserverfaces.org/

[Web03] http://www.programacion.com/articulo/introduccion_a_la_tecnologia_javaserver_faces_233

[Web04] http://www.javabeat.net/articles/19-introduction-to-ajax4jsf-1.html